L4: Entry 1 of 45 File: USPT Mar 16, 2004

DOCUMENT-IDENTIFIER: US 6708194 B1
TITLE: Porting POSIX-conforming operating systems to Win32 API-conforming operating systems

Brief Summary Text (17):
The foregoing object is attained by implementing a POSIX process by means of a Win32 process in which the primary thread of the Win32 process executes the program specified for the POSIX process and other threads are used to implement POSIX signaling, to maintain the POSIX parent-chld relationships, and to implement the semantics of the POSIX exec function.

Brief Summary Text (18):
The thread which implements POSIX signaling does so by suspending the Win32 process, modifying its state so that the Win32 process will execute the POSIX signal handler at the top of its user stack, and then resuming the Win32 process.

Detailed Description Text (39):
Win32 POSIX process 301 always has two threads: a primary thread and a signal thread. The primary thread executes the code in POSIX user program 303. The signal thread executes code in port DLL 305 which implements POSIX signal handlers. The code suspends the primary thread, examines Pproc_t 309 for the Win32 process to determine what POSIX signal must be handled and what function must handle it, adds a frame for the signal handler function to the primary thread's user stack, and then resumes the primary thread.

Detailed Description Text (45):
Other handles of interest for the present discussion include the following: thread handle 435 is the handle for the primary thread of the Win32 process which implements the POSIX process. sigevent 427 and waitevent 429 are handles of event objects. Event objects are used in Win32 operating systems to inform a thread that an event of interest to it has occurred. Typically, one thread suspends itself and waits on an event object; another thread in the Win32 process to which the first thread belongs or in another Win32 process sets the state of the event object and the Win32 operating system responds thereto by causing the waiting thread to resume execution. sigevent 427 is the event object that the signal thread waits on; depending on the POSIX signal, sigevent 427 is set by the primary thread of the Win32 process corresponding to some POSIX process or the wait thread of the Win32 process to which the handle in sigevent belongs. waitevent 429 is the event object that the wait thread waits on; waitevent 429 is created the first time the POSIX process represented by Pproc_t 309 spawns a child process and the event object is set each time the POSIX process spawns a child process.

Detailed Description Text (52):
In the preferred embodiment, a new Win32 process results whenever a POSIX process executes a fork function or a exec function. In the first case, the new Win32 process corresponds to a new POSIX process which is the child of the POSIX process executing fork; in the second case, the new Win32 process replaces the Win32 process which formerly corresponded to the POSIX process executing exec. With both fork and exec, the relationship between the POSIX process which created the new Win32 process and the new Win32 process are handled by the wait thread in the Win32

h e b b g e e e f c e h e ge

process which corresponds to the POSIX process that is executing the fork or exec function. The wait thread for the Win32 process maintains data structures which keep track of the children of the corresponding POSIX process; the wait thread further ensures that when a child POSIX process is terminated, the parent receives a SIGCHLD signal.

Detailed Description Text (71):
Continuing with the code in child_fork, the new Win32 process grows the space it needs for its copy of the user stack for the primary thread of the Win32 process which corresponds to the parent POSIX process by recursively invoking child_fork until the top of the user stack produced by the recursions is at the position in the new Win32 process's address space specified by forksp 443. forksp 443 was, however, set from the value of the stack pointer for the primary thread of the Win32 process corresponding to the parent POSIX process while that thread was executing the start_proc function invoked from the fork function. In consequence, the user stack for the primary thread of the Win32 process corresponding to the child POSIX process will be larger than that required for the Win32 process corresponding to the POSIX child process to make a copy of the primary thread user stack of the Win32 process corresponding to the POSIX parent process. The next step is to copy the read-write regions from the Win32 process corresponding to the parent POSIX process into the new Win32 process's address space. Next, duplicates arc made for the new Win32 process of those handles of the Win32 process corresponding to the POSIX parent process which were marked "close on exec". Finally, the primary thread user stack of the Win32 process corresponding to the parent is copied to the primary thread user stack of the new Win32 process.

Detailed Description Text (85):
In execve, execution proceeds with the freeing of the pproc_t entry 307 for the new Win32 process. In the preferred embodiment, freeing affects only the values of the fields pid, ppid, pgid, ntpid, and inuse of entry 307. If there is a wit thread in the Win32 process corresponding to the POSIX processing executing exec, the thread is suspended. IF the new Win32 process has already exited, the handles ph 421 and ph2423 are closed and the new Win32 process is terminated. Otherwise, execve works through Proctab 307 looking for pproc_t entries 309 for the parent of the POSIX process executing exec and the children of that POSIX process. Whenever an entry 309 is found for a Win32 process corresponding to a POSIX process which is a child of the POSIX process corresponding to the execing Win32 process, the primary thread uses the Win32 DuplicateHandle function to duplicate the handle in ph 421 for the entry from the execing Win32 process to the new Win32 process; when the entry 309 for the new Win32 process itself is found, the primary thread duplicates the handle for the new Win32 process from the new Win32 process to the parent of the execing process.

L52: Entry 1 of 2 File: USPT Jul 17, 2001

DOCUMENT-IDENTIFIER: US 6263370 B1
TITLE: TCP/IP-based client interface to network information distribution system servers

Detailed Description Text (40):
FIG. 9 displays the format for a NIDS message header. The NIDS message header comprises fourteen 8-bit bytes. Directly following the NIDS message header are the remaining data of the message that correspond either to a service request or to a response from the service to the client. The first field in the NIDS message header contains a client handle 901. Under the current implementation of the TCP/IP-based client-server interface, the client handle field is unused. The second field in the NIDS message header contain a port number 902. The port number is used by a multi-threaded client to identify the correspondence between a particular thread of the client and a request made by that thread. In essence, the port number is a thread identifier. The third field of the NIDS message header contains a message ID 903. The message ID field contains an arbitrary number selected by a client to identify a request. When a response to the request is returned to the client, the client can match the message ID in the response NIDS message header to the selected message ID that the client included in the original request in order to verify that the response matches the request. The fourth field in the NIDS header contains a destination queue field. This field is not used by the client. Instead, it is filled in by the NIDSCOM process upon receipt of the NIDS message in order to direct the db_proc process corresponding to the requested service to send a response to the request back to that same NIDSCOM process. The fifth field in the NIDS header contains a service type 905. The service type identifies the type or class of service being requested. There are a number of different categories of NIDS services, to be listed below. The sixth field 906 contains message options. Currently only the lowest bit of this 1-byte field is used. When that bit is on, the lowest bit indicates that the data following the NIDS message header has been encoded in abstract syntax notation ("ASN-1"). The seventh field in the NIDS message header is the operation code 907. This 2-byte field indicates the specific service operation requested by the application process. Each different type of NIDS service has a unique set of operation codes that correspond to that service. The service type field indicates the specific NIDS service to which the NIDS message is directed, and the operation code directs that NIDS service to execute a specific operation. The final field in a NIDS message header contains the message length 908. This is the length of the entire NIDS message including the 14 bytes in the NIDS message header.

L54: Entry 1 of 20 File: USPT Feb 24, 2004

DOCUMENT-IDENTIFIER: US 6697935 B1
TITLE: Method and apparatus for selecting thread switch events in a multithreaded processor

Abstract Text (1):
A system and method for performing computer processing operations in a data processing system includes a multithreaded processor and thread switch logic. The multithreaded processor is capable of switching between two or more threads of instructions which can be independently executed. Each thread has a corresponding state in a thread state register depending on its execution status. The thread switch logic contains a thread switch control register to store the conditions upon which a thread switch will occur. The thread switch logic has a time-out register which forces a thread switch when execution of the active thread in the multithreaded processor exceeds a programmable period of time. Thread switch logic also has a forward progress count register to prevent repetitive thread switching between threads in the multithreaded processor. Thread switch logic also is responsive to a software manager capable of changing the priority of the different threads and thus superseding thread switch events.

Parent Case Text (2):
"The present invention relates to the following U.S. applications, the subject matter of which is hereby incorporated by reference: (1) U.S. Ser. No. 08/957,002 entitled Thread Switch Control in a Multithreaded Processor System, filed concurrently herewith; (2) U.S. Pat. No. 6,105,051 entitled Apparatus and Method to Guarantee Forward progress in Execution of Threads in a Multithreaded Processor, filed concurrently herewith; (3) U.S. Pat. No. 6,212,544 entitled Altering Thread Priorities in a Multithreaded Processor, filed concurrently herewith; (4) U.S. Pat. No. 6,076,157 entitled Method and Apparatus to Force a Thread Switch in a Multithreaded Processor, filed concurrently herewith; (5) U.S. Pat. No. 6,088,788 entitled Background Completion of Instruction and Associated Fetch Request in a Multithread Processor (6) U.S. Pat. No. 6,000,011 entitled Multi-Entry Fully Associative Transition Cache; (7) U.S. Pat. No. 6,000,012 entitled Method and Apparatus for Prioritizing and Routing Commands from a Command Source to a Command Sink; (8) U.S. Pat. No. 6,035,424 entitled Method and Apparatus for Tracking Processing of a Command; (9) U.S. Pat. No. 6,049,867 entitled Method and System for Multithread Switching Only When a Cache Miss Occurs at a Second or Higher Lever; and (10) U.S. Pat. No. 5,778,243 entitled Multithreaded Cell for a Memory."

Brief Summary Text (8):
Thus, while multiple processors improve overall system performance, there are still many reasons to improve the speed of the individual CPU. If the CPU clock speed is given, it is possible to further increase the speed of the CPU, i.e., the number of operations executed per second, by increasing the average number of operations executed per clock cycle. A common architecture for high performance, single-chip microprocessors is the reduced instruction set computer (RISC) architecture characterized by a small simplified set of frequently used instructions for rapid execution, those simple operations performed quickly mentioned earlier. As semiconductor technology has advanced, the goal of RISC architecture has been to develop processors capable of executing one or more instructions on each clock cycle of the machine. Another approach to increase the average number of operations

executed per clock cycle is to modify the hardware within the CPU. This throughput measure, clock cycles per instruction, is commonly used to characterize architectures for high performance processors. Instruction pipelining and cache memories are computer architectural features that have made this achievement possible. Pipeline instruction execution allows subsequent instructions to begin execution before previously issued instructions have finished. Cache memories store frequently used and other data nearer the processor and allow instruction execution to continue, in most cases, without waiting the full access time of a main memory. Some improvement has also been demonstrated with multiple execution units with look ahead hardware for finding instructions to execute in parallel.

Brief Summary Text (10):
For both in-order and out-of-order execution in superscalar systems, pipelines will stall under certain circumstances. An instruction that is dependent upon the results of a previously dispatch ed instruction that has not yet completed may cause the pipeline to stall. For instance, instructions dependent on a load/store instruction in which the necessary data is not in the cache, i.e., a cache miss, cannot be executed until the data becomes available in the cache. Maintaining the requisite data in the cache necessary for continued execution and to sustain a high hit ratio, i.e., the number of requests for data compared to the number of times the data was readily available in the cache, is not trivial especially for computations involving large data structures. A cache miss can cause the pipelines to stall for several cycles, and the total amount of memory latency will be severe if the data is not available most of the time. Although memory devices used for main memory are becoming faster, the speed gap between such memory chips and high-end processors is becoming increasingly larger. Accordingly, a significant amount of execution time in current high-end processor designs is spent waiting for resolution of cache misses and these memory access delays use an increasing proportion of processor execution time.

Brief Summary Text (11):
And yet another technique to improve the efficiency of hardware within the CPU is to divide a processing task into independently executable sequences of instructions called threads. This technique is related to breaking a larger task into smaller tasks for independent execution by different processors except here the threads are to be executed by the same processor. When a CPU then, for any of a number of reasons, cannot continue the processing or execution of one of these threads, the CPU switches to and executes another thread. This is the subject of the invention described herein which incorporates hardware multithreading to tolerate memory latency. The term "multithreading" as defined in the computer architecture community is not the same as the software use of the term which means one task subdivided into multiple related threads. In the architecture definition, the threads may be independent. Therefore "hardware multithreading" is often used to distinguish the two uses of the term. The present invention incorporates the term multithreading to connote hardware multithreading.

Brief Summary Text (12):
Multithreading permits the processors' pipeline(s) to do useful work on different threads when a pipeline stall condition is detected for the current thread. Multithreading also permits processors implementing non-pipeline architectures to do useful work for a separate thread when a stall condition is detected for a current thread. There are two basic forms of multithreading. A traditional form is to keep N threads, or states, in the processor and interleave the threads on a cycle-by-cycle basis. This eliminates all pipeline dependencies because instructions in a single thread are separated. The other form of multithreading, and the one considered by the present invention, is to interleave the threads on some long-latency event.

Brief Summary Text (13):
Traditional forms of multithreading involves replicating the processor registers

for each thread. For instance, for a processor implementing the architecture sold under the trade name PowerPC.TM. to perform multithreading, the processor must maintain N states to run N threads. Accordingly, the following are replicated N times: general purpose registers, floating point registers, condition registers, floating point status and control register, count register, link register, exception register, save/restore registers, and special purpose registers. Additionally, the special buffers, such as a segment lookaside buffer, can be replicated or each entry can be tagged with the thread number and, if not, must be flushed on every thread switch. Also, some branch prediction mechanisms, e.g., the correlation register and the return stack, should also be replicated. Fortunately, there is no need to replicate some of the larger functions of the processor such as: level one instruction cache (L1 I-cache), level one data cache (L1 D-cache), instruction buffer, store queue, instruction dispatcher, functional or execution units, pipelines, translation lookaside buffer (TLB), and branch history table. When one thread encounters a delay, the processor rapidly switches to another thread. The execution of this thread overlaps with the memory delay on the first thread.

Brief Summary Text (14):
Existing multithreading techniques describe switching threads on a cache miss or a memory reference. A primary example of this technique may be reviewed in "Sparcle: An Evolutionary Design for Large-Scale Multiprocessors," by Agarwal et al., IEEE Micro Volume 13, No. 3, pp. 48-60, June 1993. As applied in a RISC architecture, multiple register sets normally utilized to support function calls are modified to maintain multiple threads. Eight overlapping register windows are modified to become four non-overlapping register sets, wherein each register set is a reserve for trap and message handling. This system discloses a thread switch which occurs on each first level cache miss that results in a remote memory request. While this system represents an advance in the art, modern processor designs often utilize a multiple level cache or high speed memory which is attached to the processor. The processor system utilizes some well-known algorithm to decide what portion of its main memory store will be loaded within each level of cache and thus, each time a memory reference occurs which is not present within the first level of cache the processor must attempt to obtain that memory reference from a second or higher level of cache.

Brief Summary Text (17):
An object of the present invention is to provide an improved data processing system and method for multithreaded processing embodied in the hardware of the processor. This object is achieved by a multithreaded processor capable of switching execution between two threads of instructions, and thread switch logic embodied in hardware registers with optional software override of thread switch conditions. An added advantage of the thread switch logic is that processing of various threads of instructions allows optimization of the use of the processor among the threads.

Brief Summary Text (18):
Another object of the present invention is to improve multithreaded computer processing by allowing the processor to execute a second thread of instructions thereby increasing processor utilization which is otherwise idle because it is retrieving necessary data and/or instructions from various memory elements, such as caches, memories, external I/O, direct access storage device for a first thread.

Brief Summary Text (19):
An additional object of the present invention is to provide a multithread data processing system and method which performs conditional thread switching wherein the conditions of thread switching can be different per thread or can be changed during processing by the use of a software thread control manager.

Brief Summary Text (20):
It is yet another object of the invention to provide a hardware register containing

bits embodying the events which can cause a multithreaded processor to switch threads. The feature of this invention is that the bits in this hardware register, the thread switch control register, can be enabled. This hardware register has the further advantage of improving processor performance because it is much faster than software thread switch control.

Brief Summary Text (21):
These and other related objects are achieved by providing a computer system having a multithreaded processor capable of switching processing between at least two threads of instructions when the multithreaded processor experiences one of a plurality of processor latency events. The computer system also has at least one thread state register operatively connected to the multithreaded processor, to store a state of the threads of instructions wherein the state of each thread of instructions changes when the processor switches processing to each thread. The system also has at least one thread switch control register operatively connected to the thread state register(s) and to the multithreaded processor, to store a plurality of thread switch control events which thread switch control events are enabled by setting a corresponding plurality of enable bits. The computer system further comprises a plurality of internal connections connecting the multithreaded processor to a plurality of memory elements. Access to any of the,memory elements by the multithreaded processor causes a processor latency event and the invention also has at least one external connection connecting the multithreaded processor to an external memory device, a communication device, a computer network, or an input/output device wherein access to any of the devices or the network by the multithreaded processor also causes a plurality of processor latency events. When one of the threads executing in the multithreaded processor is unable to continue execution because of one of the processor latency events and when that processor latency event is a thread switch control event whose bit is enabled, the multithreaded processor switches execution to another of the threads.

Brief Summary Text (22):
The thread switch control register has a plurality of bits, each associated uniquely with one of a plurality of thread switch control events and if one of the bits is enabled, the thread switch control event associated with that bit causes the multithreaded processor(s) to switch from one thread of instructions to another thread of instructions. The thread switch control register is programmable. Moreover, the enablement of a particular bit can be dynamically changed by either operating software or by an instruction in one of the threads.

Brief Summary Text (23):
The computer processing system may have more than one thread switch control register wherein the bit values of one thread switch control register differs from the bit values of another of said thread switch control registers.

Brief Summary Text (24):
Typically, there can be many thread switch control events, for instance, a data miss from at least one of the following: a L1-data cache, a L2 cache, storage of data that crosses a double word boundary, or an instruction miss from at least one of the following: a L1-instruction cache, a translation lookaside buffer, or a data and/or instruction miss from main memory, or an error in address translation of data and/or an instruction. Access to an I/O device external to the processor or to another processor may also be thread switch control events. Other thread switch control events comprise a forward progress count of a number of times said one of a plurality of threads has been switched from a one multithreaded processor with no instruction of the one of a plurality of threads executing, and a time-out period in which no useful work was done by the at least one processor.

Brief Summary Text (25):
The computer processing system of the invention comprises means for processing a plurality of threads of instructions; means for indicating when the processing

means stalls because one of the threads experiences a processor latency event; means for registering a plurality of thread switch control events; and means for determining if the processor latency event is one of the plurality of thread switch control events. The processing system may also comprise means for enabling the processing means to switch processing to another thread if the processor latency event is a thread switch control event.

Brief Summary Text (26):
The invention is also a method to determine the contents of a thread switch control register, comprising the steps of counting a first number of processor cycles in which a multithreaded processor is stalled because of processor latency event and counting a second number of processor cycles required for the multithreaded processor to switch processing of a first thread of instructions to a second thread of instructions, then assigning the processor latency event to be a thread switch control event by setting an enable bit in the thread switch control register if the first number is greater than the second number. Then if the enable bit is enabled, the method comprises outputting a signal to switch threads when the multithreaded processor experiences the thread switch control event if the enable bit is enabled.

Brief Summary Text (27):
The method of computer processing of the invention also comprises the steps of storing a state of a thread in a thread state register and storing a plurality of thread switch control events in a thread switch control register. Then when the state of the thread changes, a signal is output to the thread state register and the changed state of the thread is compared with the plurality of thread switch control events. If the changed state results from a thread switch control event, a signal is output to a multithreaded processor to switch execution from the thread.

Drawing Description Text (7):
FIG. 4 illustrates a block diagram of the thread switch logic, the storage control unit and the instruction unit of FIG. 2.

Drawing Description Text (8):
FIG. 5 illustrate the changes of state of a thread as the thread experiences different thread switch events shown in FIG. 4.

Detailed Description Text (6):
Instructions from the L1 I-cache 150 are preferably output to an instruction unit 220 which, in accordance with the method and system of the present invention, controls the execution of multiple threads by the various subprocessor units, e.g., branch unit 260, fixed point unit 270, storage control unit 200, and floating point unit 280 and others as specified by the architecture of the data processing system 10. In addition to the various execution units depicted within FIG. 1, those skilled in the art will appreciate that modem superscalar microprocessor systems often include multiple versions of each such execution unit which may be added without departing from the spirit and scope of the present invention. Most of these units will have as an input source operand information from various registers such as general purpose registers GPRs 272, and floating point registers FPRs 282. Additionally, multiple special purpose register SPRs 274 may be utilized. As shown in FIG. 2, the storage control unit 200 and the transition cache 210 are directly connected to general purpose registers 272 and the floating point registers 282. The general purpose registers 272 are connected to the special purpose registers 274.

Detailed Description Text (7):
Among the functional hardware units unique to this multithreaded processor 100 is the thread switch logic 400 and the transition cache 210. The thread switch logic 400 contains various registers that determine which thread will be the active or the executing thread. Thread switch logic 400 is operationally connected to the

storage control unit 200, the execution units 260, 270, and 280, and the instruction unit 220. The transition cache 210 within the storage control unit 200 must be capable of implementing multithreading. Preferably, the storage control unit 200 and the transition cache 210 permit at least one outstanding data request per thread. Thus, when a first thread is suspended in response to, for example, the occurrence of L1 D-cache miss, a second thread would be able to access the L1 D-cache 120 for data present therein. If the second thread also results in L1 D-cache miss, another data request will be issued and thus multiple data requests must be maintained within the storage control unit 200 and the transition cache 210. Preferably, transition cache 210 is the transition cache of U.S. Pat. No. 6,000,011 entitled Multi-Entry Fully Associative Transition Cache, hereby incorporated by reference. The storage control unit 200, the execution units 260, 270, and 280 and the instruction unit 220 are all operationally connected to the thread switch logic 400 which determines which thread to execute.

Detailed Description Text (11):
With respect to the multithreading capability of the processor described herein, sequencers 350 of the storage control unit 200 also output signals to thread switch logic 400 which indicate the state of data and instruction requests. So, feedback from the caches 120, 130 and 150, main memory 140, and the translation lookaside buffer 250 is routed to the sequencers 350 and is then communicated to thread switch logic 400 which may result in a thread switch, as discussed below. Note that any device wherein an event designed to cause a thread switch in a multithreaded processor occurs will be operationally connected to sequencers 350.

Detailed Description Text (12):
FIG. 4 is a logical representation and block diagram of the thread switch logic hardware 400 that determines whether a thread will be switched and, if so, what thread. Storage control unit 200 and instruction unit 220 are interconnected with thread switch logic 400. Thread switch logic 400 preferably is incorporated into the instruction unit 220 but if there are many threads the complexity of the thread switch logic 400 may increase so that the logic is external to the instruction unit 220. For ease of explanation, thread switch logic 400 is illustrated external to the instruction unit 220.

Detailed Description Text (13):
Some events which result in a thread to be switched in this embodiment are communicated on lines 470, 472, 474, 476, 478, 480, 482, 484, and 486 from the sequencers 350 of the storage control unit 200 to the thread switch logic 400. Other latency events can cause thread switching; this list is not intended to be inclusive; rather it is only representative of how the thread switching can be implemented. A request for an instruction by either the first thread T0 or the second thread T1 which is not in the instruction unit 220 is an event which can result in a thread switch, noted by 470 and 472 in FIG. 4, respectively. Line 474 indicates when the active thread, whether T0 or T1, experiences a L1 D-cache 120 miss. Cache misses of the L2 cache 130 for either thread T0 or T1 is noted at lines 476 and 478, respectively. Lines 480 and 482 are activated when data is returned for continued execution of the T0 thread or for the T1 thread, respectively. Translation lookaside buffer misses and completion of a table walk are indicated by lines 484 and 486, respectively.

Detailed Description Text (14):
These events are all fed into the thread switch logic 400 and more particularly to the thread state registers 440 and the thread switch controller 450. Thread switch logic 400 has one thread state register for each thread. In the embodiment described herein, two threads are represented so there is a T0 state register 442 for a first thread T0 and a T1 state register 444 for a second thread T1, to be described herein. Thread switch logic 400 comprises a thread switch control register 410 which controls what events will result in a thread switch. For instance, the thread switch control register 410 can block events that cause state

changes from being seen by the thread switch controller 450 so that a thread may not be switched as a result of a blocked event. The thread state registers and the logic of changing threads are the subject of a U.S. patent application Ser. No. 08/957,002, filed concurrently and herein incorporated by reference. The forward progress count register 420 is used to prevent thrashing and may be included in the thread switch control register 410. The forward progress count register 420 is the subject of U.S. Pat. No. 6,105,051, filed concurrently and herein incorporated by reference. Thread switch time-out register 430, the subject of U.S. Pat. No. 6,076,157 filed concurrently and herein incorporated by reference, allocates fairness and livelock issues. Also, thread priorities can be altered using software 460, the subject of U.S. Pat. No. 6,212,544 filed concurrently and herein incorporated by reference. Finally, but not to be limitative, the thread switch controller 450 comprises a myriad of logic gates which represents the culmination of all logic which actually determines whether a thread is switched, what thread, and under what circumstances. Each of these logic components and their functions are set forth in further detail.

Detailed Description Text (15):
Thread State Registers

Detailed Description Text (16):
Thread state registers 440 comprise a state register for each thread and, as the name suggests, store the state of the corresponding thread; in this case, a T0 thread state register 442 and a T1 thread state register 444. The number of bits and the allocation of particular bits to describe the state of each thread can be customized for a particular architecture and thread switch priority scheme. An example of the allocation of bits in the thread state registers 442, 444 for a multithreaded processor having two threads is set forth in the table below.

Detailed Description Text (17):
In the embodiment described herein, bit 0 identifies whether the miss or the reason the processor stalled execution is a result of a request for an instruction or for data. Bits 1 and 2 indicate if the requested information was not available and if so, from what hardware, i.e., whether the translated address of the data or instruction was not in the translation lookaside buffer 250, or the data or instruction itself was not in the L1 D-cache 120 or the L2 cache 130, as further explained in the description of FIG. 5. Bit 3 indicates whether the change of state of a thread results in a thread switch. A thread may change state without resulting in a thread switch. For instance, if a thread switch occurs when thread T1 experiences an L1 cache miss, then if thread T1 experiences a L2 cache miss, there will be no thread switch because the thread already switched on a L1 cache miss. The state of T1, however, still changes. Alternatively, if by choice, the thread switch logic 400 is configured or programmed not to switch on a L1 cache miss, then when a thread does experience an L1 cache miss, there will be no thread switch even though the thread changes state. Bit 8 of the thread state registers 442 and 444 is assigned to whether the information requested by a particular thread is to be loaded into the processor core or stored from the processor core into cache or main memory. Bits 15 through 17 are allocated to prevent thrashing, as discussed later with reference to the forward progress count register 420. Bits 18 and 19 can be set in the hardware or could be set by software to indicate the priority of the thread.

Detailed Description Text (18):
FIG. 5 represents four states in the present embodiment of a thread processed by the data processing system 10 and these states are stored in the thread state registers 440, bit positions 1:2. State 00 represents the "ready" state, i.e., the thread is ready for processing because all data and instructions required are available; state 10 represents the thread state wherein the execution of the thread within the processor is stalled because the thread is waiting for return of data into either the L1 D-cache 120 or the return of an instruction into the L1 I-cache

150; state 11 represents that the thread is waiting for return of data into the L2 cache 130; and the state 01 indicates that there is a miss on the translation lookaside buffer 250, i.e., the virtual address was in error or wasn't available, called a table walk. Also shown in FIG. 5 is the hierarchy of thread states wherein state 00, which indicates the thread is ready for execution, has the highest priority. Short latency events are preferably assigned a higher priority.

Detailed Description Text (19):
FIG. 5 also illustrates the change of states when data is retrieved from various sources. The normal uninterrupted execution of a thread T0 is represented in block 510 as state 00. If a L1 D-cache or I-cache miss occurs, the thread state changes to state 10, as represented in block 512, pursuant to a signal sent on line 474 (FIG. 4) from the storage control unit 200 or line 470 (FIG. 4) from the instruction unit 220, respectively. If the required data or instruction is in the L2 cache 130 and is retrieved, then normal execution of T0 resumes at block 510. Similarly block 514 of FIG. 5 represents a L2 cache miss which changes the state of thread of either T0 or T1 to state 11 when storage control unit 200 signals the miss on lines 476 or 478. (FIG. 4). When the instructions or data in the L2 cache are retrieved from main memory 140 and loaded into the processor core 100 as indicated on lines 480 and 482 (FIG. 4), the state again changes back to state 00 at block 510. The storage control unit 200 communicates to the thread registers 440 on line 484 (FIG. 4) when the virtual address for requested information is not available in the translation lookaside buffer 250, indicated as block 516, as a TLB miss or state 01. When the address does become available or if there is a data storage interrupt instruction as signaled by the storage control unit 200 on line 486 (FIG. 4), the state of the thread then returns to state 00, meaning ready for execution.

Detailed Description Text (20):
The number of states, and what each state represents is freely selectable by the computer architect. For instance, if a thread has multiple L1 cache misses, such as both a L1 I-cache miss and L1 D-cache miss, a separate state can be assigned to each type of cache miss. Alternatively, a single thread state could be assigned to represent more than one event or occurrence.

Detailed Description Text (21):
An example of a thread switch algorithm for two threads of equal priority which determines whether to switch threads is given. The algorithm can be expanded and modified accordingly for more threads and thread switch conditions according to the teachings of the invention. The interactions between the state of each thread stored in the thread state registers 440 (FIG. 4) and the priority of each thread by the thread switching algorithm are dynamically interrogated each cycle. If the active thread T0 has a L1 miss, the algorithm will cause a thread switch to the dormant thread T1 unless the dormant thread T1 is waiting for resolution of a L2 miss. If a switch did not occur and the L1 cache miss of active thread T0 turns into a L2 cache miss, the algorithm then directs the processor to switch to the dormant thread T1 regardless of the T1's state. If both threads are waiting for resolution of a L2 cache miss, the thread first having the L2 miss being resolved becomes the active thread. At every switch decision time, the action taken is optimized for the most likely case, resulting in the best performance. Note that thread switches resulting from a L2 cache miss are conditional on the state of the other thread, if not extra thread switches would occur resulting in loss of performance.

Detailed Description Text (22):
Thread Switch Control Register

Detailed Description Text (23):
In any multithreaded processor, there are latency and performance penalties associated with switching threads. In the multithreaded processor in the preferred

embodiment described herein, this latency includes the time required to complete execution of the current thread to a point where it can be interrupted and correctly restarted when it is next invoked, the time required to switch the thread-specific hardware facilities from the current thread's state to the new thread's state, and the time required to restart the new thread and begin its execution. Preferably the thread-specific hardware facilities operable with the invention include the thread state registers described above and the memory cells described in U.S. Pat. No. 6,778,243 entitled Multithreaded Storage Cell, herein incorporated by reference. In order to achieve optimal performance in a coarse grained multithreaded data processing system, the latency of an event which generates a thread switch must be greater than the performance cost associated with switching threads in a multithreaded mode, as opposed to the normal single-threaded mode.

Detailed Description Text (24):
The latency of an event used to generate a thread switch is dependent upon both hardware and software. For example, specific hardware considerations in a multithreaded processor include the speed of external SRAMs used to implement an L2 cache external to the processor chip. Fast SRAMs in the L2 cache reduce the average latency of an L1 miss while slower SRAMS increase the average latency of an L1 miss. Thus, performance is gained if one thread switch event is defined as a L1 cache miss in hardware having an external L2 cache data access latency greater than the thread switch penalty. As an example of how specific software code characteristics affect the latency of thread switch events, consider the L2 cache hit-to-miss ratio of the code, i.e., the number of times data is actually available in the L2 cache compared to the number of times data must be retrieved from main memory because data is not in the L2 cache. A high L2 hit-to-miss ratio reduces the average latency of an L1 cache miss because the L1 cache miss seldom results in a longer latency L2 miss. A low L2 hit-to-miss ratio increases the average latency of an L1 miss because more L1 misses result in longer latency L2 misses. Thus, a L1 cache miss could be disabled as a thread switch event if the executing code has a high L2 hit-to-miss ratio because the L2 cache data access latency is less than the thread switch penalty. A L1 cache miss would be enabled as a thread switch event when executing software code with a low L2 hit-to-miss ratio because the L1 cache miss is likely to turn into a longer latency L2 cache miss.

Detailed Description Text (25):
Some types of latency events are not readily detectable. For instance, in some systems the L2 cache outputs a signal to the instruction unit when a cache miss occurs. Other L2 caches, however, do not output such a signal, as in for example, if the L2 cache controller were on a separate chip from the processor and accordingly, the processor cannot readily determine a state change. In these architectures, the processor can include a cycle counter for each outstanding L1 cache miss. If the miss data has not been returned from the L2 cache after a predetermined number of cycles, the processor acts as if there had been a L2 cache miss and changes the thread's state accordingly. This algorithm is also applicable to other cases where there are more than one distinct type of latency. As an example only, for a L2 cache miss in a multiprocessor, the latency of data from main memory may be significantly different than the latency of data from another processor. These two events may be assigned different states in the thread state register. If no signal exists to distinguish the states, a counter may be used to estimate which state the thread should be in after it encounters a L2 cache miss.

Detailed Description Text (26):
The thread switch control register 410 is a software programmable register which selects the events to generate thread switching and has a separate enable bit for each defined thread switch control event. Although the embodiment described herein does not implement a separate thread switch control register 410 for each thread, separate thread switch control registers 410 for each thread could be implemented to provide more flexibility and performance at the cost of more hardware and

complexity. Moreover, the thread switch control events in one thread switch control register need not be identical to the thread switch control events in any other thread switch control register.

Detailed Description Text (27):
The thread switch control register 410 can be written by a service processor with software such as a dynamic scan communications interface disclosed in U.S. Pat. No. 5,079,725 entitled Chip Identification Method for Use with Scan Design Systems and Scan Testing Techniques or by the processor itself with software system code. The contents of the thread switch control register 410 is used by the thread switch controller 450 to enable or disable the generation of a thread switch. A value of one in the register 410 enables the thread switch control event associated with that bit to generate a thread switch. A value of zero in the thread switch control register 410 disables the thread switch control event associated with that bit from generating a thread switch. Of course, an instruction in the executing thread could disable any or all of the thread switch conditions for that particular or for other threads. The following table shows the association between thread switch events and their enable bits in the register 410.

Detailed Description Text (28):
Thread Switch Time-out Register

Detailed Description Text (29):
As discussed above, coarse grained multithreaded processors rely on long latency events to trigger thread switching. Sometimes during execution, a processor in a multiprocessor environment or a background thread in a multithreaded architecture, has ownership of a resource that can have only a single owner and another processor or active thread requires access to the resource before it can make forward progress. Examples include updating a memory page table or obtaining a task from a task dispatcher. The inability of the active thread to obtain ownership of the resource does not result in a thread switch event, nonetheless, the thread is spinning in a loop unable to do useful in work. In this case, the background thread that holds the resource does not obtain access to the processor so that it can free up the resource because it never encountered a thread switch event and does not become the active thread.

Detailed Description Text (30):
Allocating processing cycles among the threads is another concern; if software code running on a thread seldom encounters long latency switch events compared to software code running on the other threads in the same processor, that thread will get more than it's fair share of processing cycles. Yet another excessive delay that may exceed the maximum acceptable time is the latency of an inactive thread waiting to service an external interrupt within a limited period of time or some other event external to the processor. Thus, it becomes preferable to force a thread switch to the dormant thread after some time if no useful processing is being accomplished to prevent the system from hanging.

Detailed Description Text (31):
The logic to force a thread switch after a period of time is a thread switch time-out register 430 (FIG. 4), a decrementer, and a decrementer register to hold the decremented value. The thread switch time-out register 430 holds a thread switch time-out value. The thread switch time-out register 430 implementation used in this embodiment is shown in the following table:

Detailed Description Text (32):
The embodiment of the invention described herein does not implement a separate thread switch time-out register 430 for each thread, although that could be done to provide more flexibility. Similarly, if there are multiple threads, each thread need not have the same thread switch time-out value. Each time a thread switch occurs, the thread switch time-out value from the thread switch time-out register

430 is loaded by hardware into the decrement register. The decrement register is decremented once each cycle until the decrement register value equals zero, then a signal is sent to the thread switch controller 450 which forces a thread switch unless no other thread is ready to process instructions. For example, if all other threads in the system are waiting on a cache miss and are not ready to execute instructions, the thread switch controller 450 does not force a thread switch. If no other thread is ready to process instructions when the value in the decrement register reaches zero, the decremented value is frozen at zero until another thread is ready to process instructions, at which point a thread switch occurs and the decrement register is reloaded with a thread switch time-out value for that thread. Similarly, the decrement register could just as easily be named an increment register and when a thread is executing the register could increment up to some predetermined value when a thread switch would be forced.

Detailed Description Text (33):
The thread switch time-out register 430 can be written by a service processor as described above or by the processor itself with software code. The thread switch time-out value loaded into the thread switch time-out register 430 can be customized according to specific hardware configuration and/or specific software code to minimize wasted cycles resulting from unnecessary thread switching. Too high of a value in the thread switch time-out register 430 can result in reduced performance when the active thread is waiting for a resource held by another thread or if response latency for an external interrupt or some other event external to the processor is too long. Too high of a value can also prevent fairness if one thread experiences a high number of thread,switch events and the other does not. A thread switch time-out value twice to several times longer than the most frequent longest latency event that causes a thread switch is recommended, e.g., access to main memory. Forcing a thread switch after waiting the number of cycles specified in the thread switch time-out register 430 prevents system hangs due to shared resource contention, enforces fairness of processor cycle allocation between threads, and limits the maximum response latency to external interrupts and other events external to the processor.

Detailed Description Text (35):
That at least one instruction must be executed each time a thread switch occurs and a new thread becomes active is too restrictive in certain circumstances, such as when a single instruction generates multiple cache accesses and/or multiple cache misses. For example, a fetch instruction may cause an L1 I-cache 150 miss if the instruction requested is not in the cache; but when the instruction returns, required data may not be available in the L1 D-cache 120. Likewise, a miss in translation lookaside buffer 250 can also result in a data cache miss. So, if forward progress is strictly enforced, misses on subsequent accesses do not result in thread switches. A second problem is that some cache misses may require a large number of cycles to complete, during which time another thread may experience a cache miss at the same cache level which can be completed in much less time. If, when returning to the first thread, the strict forward progress is enforced, the processor is unable to switch to the thread with the shorter cache miss.

Detailed Description Text (36):
To remedy the problem of thrashing wherein each thread is locked in a repetitive cycle of switching threads without any instructions executing, there exists a forward progress count register 420 (FIG. 4) which allows up to a programmable maximum number of thread switches called the forward progress threshold value. After that maximum number of thread switches, an instruction must be completed before switching can occur again. In this way, thrashing is prevented. Forward progress count register 420 may actually be bits 30:31 in the thread switch control register 410 or a software programmable forward progress threshold register for the processor. The forward progress count logic uses bits 15:17 of the thread state registers 442, 444 that indicate the state of the threads and are allocated for the number of thread switches a thread has experienced without an instruction

executing. Preferably, then these bits comprise the forward progress counter.

Detailed Description Text (37):
When a thread changes state invoking the thread switch algorithm, if at least one instruction has completed in the active thread, the forward-progress counter for the active thread is reset and the thread switch algorithm continues to compare thread states between the threads in the processor. If no instruction has completed, the forward-progress counter value in the thread state register of the active thread is compared to the forward progress threshold value. If the counter value is not equal to the threshold value, the thread switch algorithm continues to evaluate the thread states of the threads in the processor. Then if a thread switch occurs, the forward-progress counter is incremented. If, however, the counter value is equal to the threshold value, no thread switch will occur until an instruction can execute, i.e., until forward progress occurs. Note that if the threshold register has value zero, at least one instruction must complete within the active thread before switching to another thread. If each thread switch requires three processor cycles and if there are two threads and if the thread switch logic is programmed to stop trying to switch threads after five tries; then the maximum number of cycles that the processor will thrash is thirty cycles. One of skill in the art can appreciate that there a potential conflict exists between prohibiting a thread switch because no forward progress will be made on one hand and, on the other hand, forcing a thread switch because the time-out count has been exceeded. Such a conflict can easily be resolved according to architecture and software.

Detailed Description Text (38):
FIG. 6 is a flowchart of the forward progress count feature of thread switch logic 400 which prevents thrashing. At block 610, bits 15:17 in thread state register 442 pertaining to thread T0 are reset to state 111. Execution of this thread is attempted in block 620 and the state changes to 000. If an instruction successfully executes on thread T0, the state of thread T0 returns to 111 and remains so. If, however, thread T0 cannot execute an instruction, a thread switch occurs to thread T1, or another background thread if more than two threads are permitted in the processor architecture. When a thread switch occurs away from T1 or the other background thread and execution returns to thread T0, a second attempt to execute thread T0 occurs and the state of thread T0 becomes 001 as in block 630. Again, if thread T0 encounters a thread switch event, control of the processor is switched away from thread T0 to another thread. Similarly, whenever a thread switch occurs from the other thread, e.g., T1, back to thread T0, the state of T0 changes to 010 on this third attempt to execute T0 (block 640); to 011 on the fourth attempt to execute T0 (block 650), and to state 100 on the fifth attempt to execute T0 (block 660).

Detailed Description Text (39):
In this implementation, there are five attempts to switch to thread T0. After the fifth attempt or whenever the value of bits 15:17 in the thread state register (TSR) 442 is equal to the value of bits 30:31 plus one in the thread switch control register (TSC) 410, i.e., whenever TSC(30:31)+1=TSR (15:17), no thread switch away from thread T0 occurs. It will be appreciated that five attempts is an arbitrary number; the maximum number of allowable switches with unsuccessful execution, i.e., the forward progress threshold value, is programmable and it may be realized in certain architectures that five is too many switches, and in other architectures, five is too few. In any event, the relationship between the number of times that an attempt to switch to a thread with no instructions executing must be compared with a threshold value and once that threshold value has been reached, no thread switch occurs away from that thread and the processor waits until the latency associated with that thread is resolved. In the embodiment described herein, the state of the thread represented by bits 15:17 of the thread state register 442 is compared with bits 30:31 in the thread switch control register 410. Special handling for particular events that have extremely long latency, such as interaction with input/output devices, to prevent prematurely blocking thread switching with forward

progress logic improves processor performance. One way to handle these extremely long latency events is to block the incrementing of the forward progress counter or ignore the output signal of the comparison between the forward progress counter and the threshold value if data has not returned. Another way to handle extremely long latency events is to use a separate larger forward progress count for these particular events.

Detailed Description Text (40):
Thread Switch Manager

Detailed Description Text (41):
The thread state for all software threads dispatched to the processor is preferably maintained in the thread state registers 442 and 444 of FIG. 4 as described. In a single processor one thread executes its instructions at a time and all other threads are dormant. Execution is switched from the active thread to a dormant thread when the active thread encounters a long-latency event as discussed above with respect to the forward progress register 420, the thread switch control register 410, or the thread switch time-out register 430. Independent of which thread is active, these hardware registers use conditions that do not dynamically change during the course of execution.

Detailed Description Text (42):
Flexibility to change thread switch conditions by a thread switch manager improves overall system performance. A software thread switch manager can alter the frequency of thread switching, increase execution cycles available for a critical task, and decrease the overall cycles lost because of thread switch latency. The thread switch manager can be programmed either at compile time or during execution by the operating system, e.g., a locking loop can change the frequency of thread switches; or an operating system task can be dispatched because a dormant thread in a lower priority state is waiting for an external interrupt or is otherwise ready. It may be advantageous to disallow or decrease the frequency of thread switches away from an active thread so that performance of the current instruction stream does not suffer the latencies resulting from switching into and out of it. Alternatively, a thread can forgo some or all of its execution cycles by essentially lowering its priority, and as a result, decrease the frequency of switches into it or increase the frequency of switches out of the thread to enhance overall system performance. The thread switch manager may also unconditionally force or inhibit a thread switch, or influence which thread is next selected for execution.

Detailed Description Text (43):
A multiple-priority thread switching scheme assigns a priority value to each thread to qualify the conditions that cause a switch. It may also be desirable in some cases to have the hardware alter thread priority. For instance, a low-priority thread may be waiting on some event, which when it occurs, the hardware can raise the priority of the thread to influence the response time of the thread to the event. Relative priorities between threads or the priority of a certain thread will influence the handling of such an event. The priorities of the threads can be adjusted by the thread switch manager software through the use of one or more instructions, or by hardware in response to an event. The thread switch manager alters the actions performed by the hardware thread switch logic to effectively change the relative priority of the threads.

Detailed Description Text (44):
Three priorities are used with the embodiment described herein of two threads and provides sufficient distinction between threads to allow tuning of performance without adversely affecting system performance. With three priorities, two threads can have an equal status of medium priority. The choice of three priorities for two threads is not intended to be limiting. In some architectures a "normal" state may be that one thread always has a higher priority than the other threads. It is

intended to be within the scope of the invention to cover more than two threads of execution having one or multiple priorities that can be set in hardware or programmed by software.

Detailed Description Text (45):
The three priorities of each thread are high, medium, and low. When the priority of thread T0 is the same as thread T1, there is no effect on the thread switching logic. Both threads have equal priority so neither is given an execution time advantage. When the priority of thread T0 is greater than the priority of thread T1, thread switching from T0 to T1 is disabled for all L1 cache misses, i.e., data load, data store, and instruction fetch, because L1 cache misses are resolved much faster than other conditions such as L2 misses and translates. Thread T0 is given a better chance of receiving more execution cycles than thread T1 which allows thread T0 to continue execution so long as it does not waste an excessive number of execution cycles. The processor, however, will still relinquish control to thread T1 if thread T0 experiences a relatively long execution latency. Thread switching from T1 to T0 is unaffected, except that a switch occurs when dormant thread T0 is ready in which case thread T0 preempts thread Ti. This case would be expected to occur when thread T0 switches away because of an L2 cache miss or translation request, and the condition is resolved in the background while thread T0 is executing. The case of thread T0 having a priority less than thread T1 is analogous to the case above, with the thread designation reversed.

Detailed Description Text (46):
There are different possible approaches to implementing management of thread switching by changing thread priority. New instructions can be added to the processor architecture. Existing processor instructions having side effects that have the desired actions can also be used. Several factors influence the choice among the methods of allowing software control: (a) the ease of redefining architecture to include new instructions and the effect of architecture changes on existing processors; (b) the desirability of running identical software on different versions of processors; (c) the performance tradeoffs between using new, special purpose instructions versus reusing existing instructions and defining resultant side effects; (d) the desired level of control by the software, e.g., whether the effect can be caused by every execution of some existing instruction, such as a specific load or store, or whether more control is needed, by adding an instruction to the stream to specifically cause the effect.

Detailed Description Text (47):
The architecture described herein preferably takes advantage of an unused instruction whose values do not change the architected general purpose registers of the processor; this feature is critical for retrofitting multithreading capabilities into a processor architecture. Otherwise special instructions can be coded. The instruction is a "preferred nop" or 0,0,0; other instructions, however, can effectively act as a nop. By using different versions of the or instruction, or 0,0,0 or 1,1,1 etc. to alter thread priority, the same instruction stream may execute on a processor without adverse effects such as illegal instruction interrupts. An extension uses the state of the machine state register to alter the meaning of these instructions. For example, it may be undesirable to allow a user to code some or all of these thread priority instructions and access the functions they provide. The special functions they provide may be defined to occur only in certain modes of execution, they will have no effect in other modes and will be executed normally, as a nop.

Detailed Description Text (48):
One possible implementation, using a dual-thread multithreaded processor, uses three instructions which become part of the executing software itself to change the priority of itself:

Detailed Description Text (49):

Instructions tsop 1 and tsop 2 can be the same instruction as embodied herein as or 1,1,1 but they can also be separate instructions. These instructions interact with bits 19 and 21 of the thread switch control register 410 and the problem/privilege bit of the machine state register as described herein. If bit 21 of the thread switch control register 410 has a value of one, the thread switch manager can set the priority of its thread to one of three priorities represented in the thread state register at bits 18:19. If bit 19 of the thread switch control register 410 has a value zero, then the instruction tsop 2 thread switch and thread priority setting is controlled by the problem/privilege bit of the machine state register. On the other hand, if bit 19 of the thread switch control register 410 has a value one, or if the problem/privilege bit of the machine state register has a value zero and the instruction or 1,1,1 is present in the code, the priority for the active thread is set to low and execution is immediately switched to the dormant or background thread if the dormant thread is enabled. The instruction or 2,2,2 sets the priority of the active thread to medium regardless of the value of the problem/privilege bit of the machine state register. And the instruction or 3,3,3, when the problem/privilege bit of the machine state register bit has a value of zero, sets the priority of the active thread to high. If bit 21 of the thread switch control register 320 is zero, the priority for both threads is set to medium and the effect of the or x,x,x instructions on the priority is blocked. If an external interrupt request is active, and if the corresponding thread's priority is low, that thread's priority is set to medium.

Detailed Description Text (50):
The events altered by the thread priorities are: (1) switch on L1 D-cache miss to load data; (2) switch on L1 D-cache miss for storing data; (3) switch on L1 I-cache miss on an instruction fetch; and (4) switch if the dormant thread in ready state. In addition, external interrupt activation may alter the corresponding thread's priority. The following table shows the effect of priority on conditions that cause a thread switch. A simple TSC entry in columns three and four means to use the conditions set forth in the thread switch control (TSC) register 410 to initiate a thread switch. An entry of TSC[0:2] treated as 0 means that bits 0:2 of the thread switch control register 410 are treated as if the value of those bits are zero for that thread and the other bits in the thread switch control register 410 are used as is for defining the conditions that cause thread switches. The phrase when thread T0 ready in column four means that a switch to thread T0 occurs as soon as thread T0 is no longer waiting on the miss event that caused it to be switched out. The phrase when thread T1 ready in column 3 means that a switch to thread T1 occurs as soon as thread T1 is no longer waiting on the miss event that caused it to be switched out. If the miss event is a thread switch time-out, there is no guarantee that the lower priority thread completes an instruction before the higher priority thread switches back in.

Detailed Description Text (51):
It is recommended that a thread doing no productive work be given low priority to avoid a loss in performance even if every instruction in the idle loop causes a thread switch. Yet, it is still important to allow hardware to alter thread priority if an external interrupt is requested to a thread set at low priority. In this case the thread is raised to medium priority, to allow a quicker response to the interrupt. This allows a thread waiting on an external event to set itself at low priority, where it will stay until the event is signalled.

Detailed Description Paragraph Table (1):
Thread State Register Bit Allocation (0) Instruction/data 0 = Instruction 1 = Data (1:2) Miss type sequencer 00 = None 01 = Translation lookaside buffer miss (check bit 0 for I/D) 10 = L1 cache miss 11 = L2 cache miss (3) Transition 0 = Transition to current state does not result in thread switch 1 = Transition to current state results in thread switch (4:7) Reserved (8) 0 = Load 1 = Store (9:14) Reserved (15:17) Forward progress counter 111 = Reset (instruction has completed during this thread) 000 = 1st execution of this thread w/o instruction complete 001 = 2nd

execution of this thread w/o instruction complete 010 = 3rd execution of this thread w/o instruction complete 011 = 4th execution of this thread w/o instruction complete 100 = 5th execution of this thread w/o instruction complete (18:19) Priority (could be set by software) 00 = Medium 01 = Low 10 = High 11 = <Illegal> (20:31) Reserved (32:63) Reserved if 64 bit implementation

Detailed Description Paragraph Table (2):
Thread Switch Control Register Bit Assignment (0) Switch on L1 data cache fetch miss (1) Switch on L1 data cache store miss (2) Switch on L1 instruction cache miss (3) Switch on instruction TLB miss (4) Switch L2 cache fetch miss (5) Switch on L2 cache store miss (6) Switch on L2 instruction cache miss (7) Switch on data TLB/segment lookaside buffer miss (8) Switch on L2 cache miss and dormant thread not L2 cache miss (9) Switch when thread switch time-out value reached (10) Switch when L2 cache data returned (11) Switch on IO external accesses (12) Switch on double-X store: miss on first of two* (13) Switch on double-X store: miss on second of two* (14) Switch on store multiple/string: miss on any access (15) Switch on load multiple/string: miss on any access (16) Reserved (17) Switch on double-X load: miss on first of two* (18) Switch on double-X load: miss on second of two* (19) Switch on or 1,1,1 instruction if machine state register (problem state) bit, msr(pr) = 1. Allows software priority change independent of msr(pr). If bit 19 is one, or 1,1,1 instruction sets low priority. If bit 19 is zero, priority is set to low only if msr(pr) = 0 when the or 1,1,1 instruction is executed. See changing priority with software, to be discussed later. (20) Reserved (21) Thread switch priority enable (22:29) Reserved (30:31) Forward progress count (32:63) Reserved in 64 bit register implementation *A double-X load/store refers to loading or storing an elementary halfword, a word, or a double word, that crosses a doubleword boundary. A double-X load/store in this context is not a load or store of multiple words or a string of words.

Detailed Description Paragraph Table (3):
Thread Switch Time-out Register Bits (0:21) Reserved (22:31) Thread switch time-out value

Detailed Description Paragraph Table (4):
tsop 1 or 1,1,1 - Switch to dormant thread tsop 2 or 1,1,1 - Set active thread to LOW priority - Switch to dormant thread - NOTE: Only valid in privileged mode unless TSC[19]=1 tsop 3 or 2,2,2 - Set active thread to MEDIUM priority tsop 4 or 3,3,3 - Set active thread to HIGH priority - NOTE: Only valid in privileged mode

Detailed Description Paragraph Table (5):
T0 Thread T1 Thread T0 T1 Switch Switch Priority Priority Conditions Conditions High High TSC TSC High Medium TSC[0:2] treated as 0 TSC or if T0 ready High Low TSC [0:2] treated as 0 TSC or if T0 ready Medium High TSC or if T1 ready TSC[0:2] treated as 0 Medium Medium TSC TSC Medium Low TSC[0:2] treated as 0 TSC or if T0 ready Low High TSC or if T1 ready TSC[0:2] treated as 0 Low Medium TSC or if T1 ready TSC[0:2] treated as 0 Low Low TSC TSC

Other Reference Publication (4):
Elkateeb, Ali et al; IEEE Pacific Rim Conference 1993, pp. 141-144, "A Task Allocation by Priority Strategy for RISC Architecture Supported with Non-Overlapped Multiple Register Set: A Complexity Study ".

Other Reference Publication (5):
Fiske, Stuart et al; Future Generation Computer Systems II, Oct. 1995, No. 6, pp. 503-518, "Thread Prioritization: A Thread Scheduling Mechanism for Multiple-Contett Parallel Processors".

Other Reference Publication (6):
IBM Technical Disclosure Bulletin, vol. 38, No. 5, May 1995, pp. 271-276, "Deterministic Priority Inversion Method for Personal Computers".

Other Reference Publication (9):
Weinberg, William; Posix, Real-Time Magazine 97-2, pp. 51-54, "Meeting Real-Time Performance Goals With Kernel Threads".

Other Reference Publication (11):
Inohara, Shigekazu et al; IEEE, 1993, pp. 149-155, "Unstable Threads Kernel Interface for Minimizing the Overhead of Thread Switching".

Other Reference Publication (14):
Tokuda, Hideyuki et al; IEEE, 1989, pp. 348-359, "Priority Inversions in Real-Time Communication".

Other Reference Publication (54):
Hidaka, Yasuo et al, "Multiple Threads in Cyclic Register Windows", Proceedings of the 20th Annual Int'l. Symposium on Computer Architecture, May 193, PP. 131-142.

Other Reference Publication (74):
Bradley J. Kish and Bruno R. Preiss, "Hobbes: A Multi-Threaded Supercalar Architecture," University of Waterloo, 1994.

CLAIMS:

1. A computer processor comprising: (a) at least one multithreaded processor to switch execution between a plurality of threads of instructions; and (b) at least one thread switch control register having a plurality of bits, each of said bits associated uniquely with one of a plurality of thread switch control events.

2. The processor of claim 1 wherein if one of the bits is enabled, the thread switch control event associated with that bit causes the at least one multithreaded processor to switch from one of a plurality of threads to another of said plurality of threads.

3. The processor of claim 2 comprising more than one thread switch control register.

4. The processor of claim 3 wherein the bit values of one thread switch control register differs from the bit values of another of said thread switch control registers.

5. The processor of claim 2 wherein the plurality of thread switch control events comprise a forward progress count of a number of times the one of a plurality of threads has been switched from the at least one multithreaded processor with no instruction of the one of a plurality of threads executing.

6. The processor of claim 2 wherein the plurality of thread switch control events comprise a time-out period.

7. The processor of claim 1 wherein the thread switch control register is programmable.

8. The processor of claim 7 wherein at least one instruction can disable at least one of the bits in the thread switch control register.

9. The processor of claim 1 wherein the plurality of thread switch control events comprise a data miss from at least one of the following: a L1-data cache, a L2 cache, a translation lookaside buffer.

10. The processor of claim 1 wherein the plurality of thread switch control events comprise an instruction miss from at least one of the following: a L1-instruction

cache, a translation lookaside buffer.

11. The processor of claim 1 wherein the plurality of thread switch control events comprise an error in address translation of data and/or an instruction.

12. The processor of claim 1 wherein the plurality of thread switch control events comprise access to an I/O device external to said processor.

13. The processor of claim 1 wherein the plurality of thread switch control events comprise access to another processor.

14. A computer processing system, comprising: (a) a multithreaded processor which experiences a plurality of thread switch control events; (b) a thread switch control register interconnected with the multithreaded processor wherein the thread switch control register has a plurality of enable bits each uniquely associated with one of a plurality of thread switch control events comprising: a data store/load cache miss, an instruction cache miss, a data/instruction address translation miss, access to another processor, access to an external I/O device, a time-out period, a threshold count not to exceed a number of thread switches without an instruction executing, wherein the multithreaded processor experiences one of the thread switch control events and if the experienced thread switch, control event is associated with one of the enable bits and if the associated enable bit is enabled, then the multithreaded processor will switch threads.

15. A computer processing system comprising: (a) means for processing a plurality of threads of instructions; (b) means for indicating when the processing means stalls because one of the plurality of threads experiences a processor latency event; (c) means for registering a plurality of thread switch control events; and (d) means for determining if the processor latency event is one of the plurality of thread switch control events.

16. The computer processing system of claim 15, further comprising: (e) means for enabling the processing means to switch processing to another of the plurality of threads if the processor latency event is one of the plurality of thread switch control events.

17. A method of computer processing comprising the steps of: (a) storing a state of a thread in a thread state register; (b) storing a plurality of thread switch control events in a thread switch control register; (c) signaling the thread state register when the state of the thread changes; (d) comparing the changed state of the thread with the plurality of thread switch control events.

18. The method of claim 17, further comprising: (e) signalling a multithreaded processor to switch execution from the thread if the changed state results from a thread switch control event.

19. A computer system, comprising: (a) a multithreaded processor capable of switching processing between at least two threads of instructions when the multithreaded processor experiences one of a plurality of processor latency events; (b) at least one thread state register operatively connected to the multithreaded processor to store a state of the threads of instructions wherein the state of each thread of instructions changes when the processor switches processing to each thread; (c) at least one thread state control register operatively connected to the at least one thread state register and to the multithreaded processor, to store a plurality of thread switch control events which thread switch control events are enabled by setting a corresponding plurality of enable bits; (d) a plurality of internal connections connecting the multithreaded processor to a plurality of memory elements wherein access to any of the plurality of memory elements by the multithreaded processor causes a processor latency event; wherein when one of the threads executing in the multithreaded processor is unable to continue execution

because of one of the processor latency events and when that processor latency event is a thread switch control event and when that corresponding enable bit is set, the multithreaded processor switches execution to another of the threads.

21. A computer processor comprising: (a) at least one multithreaded processor to switch execution between a plurality of threads of instructions; and (b) at least one programmable thread switch control register having a plurality of bits, each of said bits associated uniquely with one of a plurality of thread switch control events wherein if one of the bits is enabled, the thread switch control event associated with that bit causes the at least one multithreaded processor to switch from one of a plurality of threads to another of said plurality of threads and wherein the plurality of thread switch control events comprise a data miss from at least one of the following: a L1-data cache, a L2 cache, a translation lookaside buffer; an instruction miss from at least one of the following: a L1-instruction cache, a translation lookaside buffer; an error in address translation of data and/or an instruction; access to an I/O external to said processor; access to another processor; a forward progress count of a number of times said one of a plurality of threads has been switched from the at least one multithreaded processor with no instruction of the one of a plurality of threads executing; and a time-out period.

(2)

First Hit Fwd Refs

☐ Generate Collection Print

L16: Entry 5 of 10 File: USPT Aug 29, 2000

DOCUMENT-IDENTIFIER: US 6112222 A
TITLE: Method for resource lock/unlock capability in multithreaded computer environment

Abstract Text (2):
multiple threads concurrently desire the lock. This alternate process employs at least one function in the POSIX threads standard to implement a queue of waiting threads. A similar hybrid approach to the unlock capability is also provided.

Brief Summary Text (3):
"SYSTEM FOR RESOURCE LOCK/UNLOCK CAPABILITY IN MULTITHREADED COMPUTER ENVIRONMENT," by Govindaraju et al., Ser. No. 09/139,255, (Docket No. PO9-98-178); and

Brief Summary Text (4):
"RESOURCE LOCK/UNLOCK CAPABILITY IN MULTITHREADED COMPUTER ENVIRONMENT," by Govindaraju et al., Ser. No. 09/138,996, (Docket No. PO9-98-179).

Brief Summary Text (6):
This invention relates to capabilities for managing shared resources in a computer system, and, more particularly, to shared resource management techniques in multithread environments.

Brief Summary Text (10):
A thread standard has now been incorporated into the POSIX standard. Basic thread management under the POSIX standard is described, for example, in a publication by K. Robbins and S. Robbins entitled Practical UNIX Programming--A Guide To Concurrency, Communication and Multi-threading, Prentice Hall PTR (1996).

Brief Summary Text (12):
Two types of locks are often encountered in the art, namely blocking locks and simple or "spin" locks. Blocking locks are of the form which cause a thread requesting the lock to cease being executable, e.g., to go to "sleep" as the term is employed in the art, if the lock is currently held by another thread. Spin locks, in contrast, do not put waiting threads to "sleep", but rather, the waiting threads execute a spin loop, and thus repeatedly continue to request the lock until it is freed by the current thread "owner". Blocking locks are typically used for large critical sections of code or if the operating system kernel must differentiate between threads requiring data structure read-only capability and threads requiring the capability to modify the data structure(s).

Brief Summary Text (13):
One other term to note is the concept of code being multithread-safe. Code is considered to be thread/MP-safe if multiple execution threads contending for the same resource or routine are serialized such that data integrity is insured for all threads. One way of effecting this is by means of the aforementioned locks.

Brief Summary Text (14):
By way of further background, one approach to shared and exclusive access control in a multiprocessor system is presented in U.S. Pat. No. 4,604,694, entitled "Shared and Exclusive Access Control". Briefly described, this patent employs a

lockword to control access to a queue of the resource desired and indicates both the present use of the resource and a pointer to the most recently enqueued task in the queue. Methods using an atomic, double compare and swap operation allow a task requesting either exclusive or shared access of the resource to be enqueued, and allow tasks requiring either exclusive or shared access to the resource to suitably rearrange the queue and prepare access to the resource for other tasks. The approach is hardware dependent in that the method relies on the atomic double compare and swap operation of, for example, an IBM System/370 product. Unfortunately, many of today's multi-processing systems, such as an RS/6000 system offered by International Business Machines Corporation, lack this particular instruction capability.

Brief Summary Text (15):
Presently, thread locking employs standard POSIX mutex functions. These standard POSIX functions include pthread.sub.-- mutex.sub.-- lock and pthread.sub.-- mutex.sub.-- unlock which are described, for example, in the above-referenced publication by K. Robbins & S. Robbins entitled Practical UNIX Programming--A Guide to Concurrency, Communication and Multi-threading. These functions are designed to enhance portability of applications running on several operating systems. Unfortunately, the functions have the disadvantage of poor performance and are often inefficient for high performance libraries, such as a threaded message passing interface (MPI) library, particularly, since uncontested performance is the most important marketing and evaluation criterion.

Brief Summary Text (16):
Thus, a need exists in the art for a commercially enhanced approach to multithread-safe resource locking and unlocking in a multithread computer environment.

Brief Summary Text (18):
Briefly summarized, the invention comprises in one aspect a method for obtaining a lock on a resource in a multithread computer environment. The method includes: determining whether one thread or multiple threads desire the resource lock; directly assigning resource ownership to the one thread when the determining determines that only the one thread is actively seeking a lock on the resource, the directly assigning employing a first lock process which comprises one of an operating system primitive lock process or a hardware lock process; and, employing a second lock process to obtain the lock on the resource when the determining determines that multiple threads concurrently desire ownership of the resource, the second lock process employing at least one function in the POSIX threads standard.

Brief Summary Text (19):
In another aspect, a method for unlocking a lock on a resource in a multithread computer environment is provided. The lock employs a lock structure including a lock.owner field representative of thread ownership of the resource, a lock.status field representative of ownership of the lock structure, and a lock.waiters field representative of a count of threads waiting to obtain the lock. The method includes: obtaining control of the lock structure by setting the lock.status field; determining whether any threads are waiting for the lock by evaluating the lock.waiters field; and if no threads are waiting, directly setting the lock.owner field to null, otherwise employing at least one function in the POSIX threads standard to set the lock.owner field to null and issue a thread condition signal to waiting threads that the resource has been unlocked.

Brief Summary Text (20):
To restate, a hybrid lock function (and/or macro) is presented herein which has minimal impact on performance when only a single thread is active, but which provides correct operation using mutex locks when multiple threads are active. When lock is to be acquired, the lock state is tested via AIX-provided atomic test functions. If the lock is unowned and if there are no waiters, the lock is claimed by the thread and ownership set via AIX-provided atomic functions. These have

minimal overhead, and correspond to the case in which only one thread is trying to acquire the lock. However, if the lock is owned and/or there are already threads waiting to acquire the lock, the thread updates the wait count and does a POSIX thread condition wait, thus putting itself to sleep awaiting an unlock signal. When the current lock is released, a similar set of tests is performed by the releasor. If there are no waiters, the global ownership variable is atomically reset; otherwise, a POSIX thread signal is sent to awaken a waiting thread. A similar hybrid approach to the unlock function is also presented herein.

Detailed Description Text (4):
As shown, a computer environment 100 includes a plurality of computing nodes 102 coupled to one another via a connection 104. As one example, each computing node may comprise a node of an RS/6000 SP System offered by International Business Machines Corporation, and connection 104 may be a packet switch network, such as the SP switch or high performance switch (HPS), also offered by International Business Machines Corporation. Note again, FIG. 1 is presented by way of example only. The techniques disclosed herein could apply to any serial program or any multithreaded program running on a single machine in addition to the multiprocessor environment depicted in FIG. 1.

Detailed Description Text (5):
Within environment 100, message packets are passed from a source computing node (sender) to a receiver computing node (receiver) via packet switch network 104. For example, a user task 106 of computing unit N may pass a message to a user task 106 of computing unit 1 (receiver). Each user task can directly read data from and write data to an associated adapter 112, bypassing the overhead normally associated with having the operating system intervene in communication protocols. Adapter 112 couples computing unit 102 to switch 104. One example of switch 104 is described in detail in "IBM Parallel System Support Programs For AIX Administration Guide," Publication No. GC23-3897-02 (1996).

Detailed Description Text (6):
As further explanation, communication between a computing unit and its associated adapter 112 is, for instance, described by an interface that includes functions, such as, open communication, close communication, enable route, disable route, return status, and reset adapter. In one embodiment, the interface comprises a message passing interface (MPI) 110, also referred to herein as an MPI library. The MPI library comprises one example of a resource for which a lock mechanism in accordance with the present invention may be employed.

Detailed Description Text (14):
.waiter(s) (count of threads waiting on this lock)

Detailed Description Text (17):
In accordance with this invention, the lock.status field indicates whether the control structure itself is held (or "set"), or is "free". As explained further below, a lock call must first ensure possession of the lock structure before attempting to evaluate and obtain the lock condition on the resource. The lock.owner field identifies whether a thread has a lock on the resource, and if so, the owner ID, while the lock.waiter field is a count of the threads waiting for a lock on the resource.

Detailed Description Text (18):
The lock.mutex and lock.cond structures are employed by functions of the POSIX threads standard as shown in flowcharts FIGS. 3a-4. The lock.mutex field is used by a "pthread" type lock in accordance with one branch of processing pursuant to the present invention, while the lock.cond structure identifies a thread in wait condition that will ultimately

Detailed Description Text (20):

h e b b g e e e f c e e ge

FIGS. 3a & 3b present one embodiment of a hybrid lock process in accordance with the present invention. A lock call 300, which references an associated lock structure ("lock"), and includes a thread ID ("me"), initially determines whether the status field of the lock structure is "free" 302. If "no", then processing waits for the status field to be free (e.g., using a short wait instruction). Once lock.status is free, the status is "set" 304, meaning that the lock structure is then owned by the calling thread. Inquiry 302 and instruction 304 essentially comprise an atomic compare & swap function which is a function provided as a direct kernel call (e.g., .sub.-- check.sub.-- lock) by AIX and is thus implemented with minimum overhead.

Detailed Description Text (21):
Once the lock structure is "set", processing determines whether there are any threads waiting (.waiters) for a lock on the resource and whether there is a current lock (.owner) on the resource. If both lock.waiters is 0 and lock.owner is a null, then the current lock can be directly assigned by setting lock.owner equal to "me" 308, i.e., the calling thread's ID. This transition from inquiry 306 to instruction 308 is an optimized lock path in accordance with the invention. Essentially, since only one thread desires a lock on the resource, a truncated lock approach is employed. This truncated process can comprise one of an operating system primitive lock or a hardware lock process. For example, an operating system primitive lock function may comprise the .sub.-- check.sub.-- lock() function to acquire the lock status, and the .sub.-- clear.sub.-- lock() function to release the lock status, while the hardware process may comprise a hardware compare and swap instruction or an atomic store instruction. After setting the owner field, lock.status is set "free" 310 and processing returns from the lock call 312. Freeing lock.status can again comprise an AIX provided atomic function such as the "clear.sub.-- lock" function.

Detailed Description Text (22):
If there are threads waiting for a resource lock or there is currently a lock on the resource, then processing increments the lock.waiters count 314 and returns lock.status to "free" 316. This alternate lock branch builds a queue of waiters for the lock, represented by a list of threads waiting on the thread unlock condition (lock.cond). This approach employs functions in the POSIX threads standard, including a pthread.sub.-- mutex.sub.-- lock function, a pthread.sub.-- condition.sub.-- wait function and a pthread.sub.-- mutex.sub.-- unlock function. The pthread.sub.-- mutex.sub.-- lock function is a mutually exclusive lock described in greater detail in the above-incorporated materials entitled "Technical Reference, Volumes 1 & 2, Base Operating System & Extensions," Version 4, 4th Edition (October 1996). The pthread.sub.-- mutex.sub.-- lock function is employed using the lock.mutex field 318.

Detailed Description Text (23):
Processing next determines whether lock.owner is other than "null" 320. Assuming that there is a current owner, then the pthread.sub.-- cond.sub.-- wait function is implemented using the lock.cond variable address and lock.mutex address 322. Essentially, since another thread owns the lock, the current thread calling for the lock is placed in a wait state until, e.g., it receives a wakeup signal. After receiving the wakeup signal, the thread will try again to determine whether the control lock is free. Once all prior lock requests have been satisfied, then lock.owner is set to the current thread's ID ("me") 324 and the pthread.sub.-- mutex is unlocked 326.

Detailed Description Text (25):
One embodiment of an unlock call 400 in accordance with the present invention is depicted in FIG. 4. In this embodiment, the unlock call again provides certain "lock" information and a thread ID ("me"). The atomic compare and swap of lock.status from "free" to "set" is initially performed via inquiry 402 and instruction 404 as discussed above. Upon setting the lock.status field, authority

to work with the variables of the lock structure is granted to the unlock call. Thereafter, processing determines whether any thread is waiting on a lock 406. If "no", then the lock.owner field is updated to null 408.

Detailed Description Text (26):
However, if one or more threads is waiting on the lock, a pthread.sub.-- mutex.sub.-- lock is employed in order to allow communication to the other waiting threads 410. After obtaining this pthread lock, the lock.owner is set to null 412 and the pthread.sub.-- cond.sub.-- signal function is employed to send a lock release signal to the waiting threads 414. After sending this thread condition signal, the pthread.sub.-- mutex is unlocked and the waiting thread(s) is clear to proceed 416. After unlocking 416 or setting lock.owner to null 408, the lock.status field is freed 418 and processing returns from the unlock call 420.

Other Reference Publication (3):
K.A. Robbins and S. Robbins, "Practical UNIX Programming: A Guide to Concurrency, Communication, and Multithreading", p. 10, pp. 333-334, 347-349 & 365-369, (1996).

CLAIMS:

1. A method for obtaining a lock on a resource in a multithread computer environment, said method comprising:

determining whether one thread or multiple threads desire said lock on said resource;

directly assigning resource ownership to said one thread when said determining determines only said one thread to be actively seeking said lock on said resource, said directly assigning employing a first lock process comprising one of an operating system primitive lock process or a hardware lock process; and

employing a second lock process to obtain said lock on said resource when said determining determines that multiple threads concurrently desire ownership of said resource, said second lock process employing at least one function in the POSIX threads standard.

4. The method of claim 3, wherein said lock structure further includes a lock.waiter field representative of a number of threads waiting for said lock, and wherein said determining comprises evaluating said lock.waiters field and said lock.owner field to ascertain whether only said one thread actively desires said lock on said resource.

5. The method of claim 4, wherein said second lock process comprises incrementing said lock.waiters, releasing said lock.status, and implementing a queue of waiting threads when said determining determines that multiple threads concurrently desire ownership of said resource.

6. The method of claim 5, wherein said implementing said queue of waiting threads comprises employing a mutex function and a condition wait function of the POSIX threads standard.

7. The method of claim 6, further comprising allocating at initialization of a user of said multithread computer environment a "mutex" and a "cond" structure for use in implementing said mutex function and said wait condition function.

9. The method of claim 1, wherein said second lock process employs said at least one function in the POSIX threads standard to implement a queue of threads waiting for said lock on said resource, said at least one function in the POSIX threads standard comprising a mutex function and a condition wait function.

11. The method of claim 1, wherein said resource comprises a threaded message passing interface (MPI) library.

14. A method for unlocking a lock on a resource in a multithread computer environment, said lock employing a lock structure including a lock.owner field representative of thread ownership of said resource, a lock.status field representative of ownership of said lock structure, and a lock.waiters field representative of a count of threads waiting to obtain said lock, said method comprising:

obtaining control of said lock structure by setting said lock.status field;

determining whether any threads are waiting for said lock by evaluating said lock.waiters field; and

if no threads are waiting, directly setting said lock.owner field to null, otherwise employing at least one function in the POSIX threads standard to set said lock.owner field to null and issue a thread condition signal to waiting threads that said resource has been unlocked.

15. The method of claim 14, wherein said at least one function in the POSIX threads standard comprises a mutex function and a thread condition signal function, and wherein said mutex function employs a "mutex" structure and said thread condition signal employs a "cond" structure, said "mutex" structure and said "cond" structure being provided by a user of said multithread computer environment.

17. The method of claim 14, wherein said multithread computer environment comprises an AIX based multi-processor system.

Hit List

Clear | Generate Collection | Print | Fwd Refs | Bkwd Refs | Generate OACS

Search Results - Record(s) 1 through 10 of 10 returned.

☐ 1. Document ID: US 6457064 B1

L16: Entry 1 of 10 File: USPT Sep 24, 2002

US-PAT-NO: 6457064
DOCUMENT-IDENTIFIER: US 6457064 B1

TITLE: Method and apparatus for detecting input directed to a thread in a multi-threaded process

DATE-ISSUED: September 24, 2002

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Huff; Daryl A.	Saratoga	CA		
Yeager; William J.	Menlo Park	CA		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Sun Microsystems, Inc.	Palo Alto	CA			02

APPL-NO: 09/ 067546 [PALM]
DATE FILED: April 27, 1998

PARENT-CASE:
This application is related to U.S. Patent Application Nos. (not yet assigned). (Attorney Docket No. P2668/SUN1P169), filed on the same date herewith and commonly assigned, entitled "HIGH PERFORMANCE MESSAGE STORE," (Attorney Docket No. P2945/SUN1P192), filed on the same date herewith and commonly assigned, entitled "METHOD AND APPARATUS FOR HIGH PERFORMANCE ACCESS TO DATA IN A MESSAGE STORE", (Attorney Docket No. P3088/SUN1P194), filed on the same date herewith and commonly assigned, entitled "CRITICAL SIGNAL THREAD," which are incorporated herein by reference.

INT-CL: [07] G06 F 9/54

US-CL-ISSUED: 709/318; 709/100
US-CL-CURRENT: 719/318; 718/100

FIELD-OF-SEARCH: 709/100-108, 709/310-400, 709/102, 709/104, 709/318

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

h e b b g e e e f e ef b e

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
5179702	January 1993	Spix et al.	709/102
5247675	September 1993	Farrell et al.	709/103
5257372	October 1993	Furtney et al.	709/105
5287508	February 1994	Hejna, Jr. et al.	709/102
5305455	April 1994	Anschuetz et al.	709/100
5339415	August 1994	Strout, II et al.	709/102
5355488	October 1994	Cox et al.	709/100
5907702	May 1999	Flynn et al.	709/108
5933627	August 1999	Parady	712/228
6052708	April 2000	Flynn et al.	709/108
6085215	July 2000	Ramakrishnan et al.	709/102

FOREIGN PATENT DOCUMENTS

FOREIGN-PAT-NO	PUBN-DATE	COUNTRY	US-CL
0 817 047	January 1998	EP	
0 817 047	July 1998	EP	

OTHER PUBLICATIONS

Crispin, M., "Internet Message Access Protocol," University of Washington, (1996), pp. 1-93.
Lawrence, K.: Writing Multithreaded Graphics Programs, The Developer Connection News, vol. 7, Apr. 1995 (1995-04), pp. 16-17, XP002188186.
Kleiman, S., et al.: Writing Multithreaded Code In Solaris, Intellectual Leverage. San Francisco, Feb. 24-28, 1992, Proceedings of the Computer Society International Conference (COMPCON) Spring, Los Alamitos, IEEE Comp. Soc. Press, US, vol. Conf. 37, (Feb. 24, 1992), pp. 187-192, XP010027136.
Evans, S.: The Notifier, USENIX Association Summer Conference Proceedings, Atlanta 1986, Jun. 9-13, 1986, pp. 344-354, XP002188187, El Cerrito, Calif. USA, USENIX Assoc., USA.
Event-Raising Mechanism for Networking Architecture, IBM Technical Disclosure Bulletin, IBM Corp., New York, US, vol. 39, No. 1, 1996, pp. 381-384, XP000556437, ISSN: 0018-8689.
INNOSOFT INTERNATIONAL: "PMDF System Manager's Guide--Chapter 11" PMDF DOCUMENTATION, `Onfine!` No. PMDF-REF-5.1, Aug. 1997, pp. 1-17, XP002187544, Retrieved from the Internet: URL:http://ruls01.fsw.LeidenUniv.nl/ {Versteegen/pmdf/DOC/HTML/sysman/ book_b.html, retrieved on Jan. 16, 2002.

ART-UNIT: 2151

PRIMARY-EXAMINER: Courtenay, III; St. John

ASSISTANT-EXAMINER: Nguyen; Van H.

ATTY-AGENT-FIRM: Beyer Weaver & Thomas LLP

ABSTRACT:

A method and apparatus are disclosed for handling an input event directed to a

h e b b g e e e f e ef b e

L16: Entry 1 of 10 File: USPT Sep 24, 2002

DOCUMENT-IDENTIFIER: US 6457064 B1
TITLE: Method and apparatus for detecting input directed to a thread in a multi-threaded process

Abstract Text (1):
A method and apparatus are disclosed for handling an input event directed to a thread within a process operating in a multi-threaded system. A process is alerted that an input event effecting one of its active connection threads has been received. An input polling thread in the process is enabled and is used, in conjunction with other thread-specific data, to determine which of the threads in the process has an event directed to it. That thread is then triggered to handle the input event. The active connection thread receiving the input event is assigned a light weight process to execute only after it is determined that the thread requires it to process the input event. The input polling thread for a process detects input events for its process and causes the appropriate connection thread in the process to be assigned a light weight process when the connection thread needs it to execute. This greatly reduces the number of light weight processes assigned to threads in a multi-threaded operating system.

Parent Case Text (1):
This application is related to U.S. Patent Application Nos. (not yet assigned). (Attorney Docket No. P2668/SUN1P169), filed on the same date herewith and commonly assigned, entitled "HIGH PERFORMANCE MESSAGE STORE," (Attorney Docket No. P2945/SUN1P192), filed on the same date herewith and commonly assigned, entitled "METHOD AND APPARATUS FOR HIGH PERFORMANCE ACCESS TO DATA IN A MESSAGE STORE", (Attorney Docket No. P3088/SUN1P194), filed on the same date herewith and commonly assigned, entitled "CRITICAL SIGNAL THREAD," which are incorporated herein by reference.

Brief Summary Text (3):
The present invention relates generally to the field of computer software and client/server applications. In particular, it relates to directing data from clients to appropriate processes and threads in a multi-threaded operating system.

Brief Summary Text (8):
In some cases, a process, or threads within a process, may be waiting on an external event to occur or for input from an external source. For the process or thread to be "listening" for some type of external signal or data, it requires that it have a light weight process behind it. If the number of threads waiting for input or the number of processes running keeps growing, the more light weight processes the operating system will need to assign. When all the light weight processes have been assigned and a process P1 is ready to execute (and, thus, needs a light weight process), some operating systems perform context switching with another process, for example, P2. Context switching is a comparatively resource-intensive operation where the operating system to accommodate the needs of P1, saves the context of P2, restores the context of P1 and assigns to it the now available light weight process from P2. The system then allows P1 to run and, when it is complete or remains idle, restores the context of P2 and assigns the light weight process. This type of context switching (i.e. saving and restoring processes' contexts) for entire processes normally requires significant processing

time and system resources and, thus, is undesirable. The concept of having threads within a process stemmed from the desire to reduce process context switching.

Brief Summary Text (9):
It is also undesirable to have each thread, especially in high-volume multi-threaded operating systems, assigned its own exclusive light weight process. This situation is particularly inefficient and wasteful given that, in many situations, threads are idle or in some type of input wait state most of the time, where they are not actively processing data or executing. In a high-volume network with thousands of users, the number of light weight processes can begin to run low. This problem is also true in smaller networks where the number of light weight processes available in the operating system may have been initially set to be relatively small. Even though starting and stopping a light weight process does not require significant overhead, it is very desirable for the operating system to always have them available when needed. Regardless of the size of the computing environment, threads can easily proliferate and consume valuable system resources, such as light weight processes.

Brief Summary Text (12):
To achieve the foregoing, and in accordance with the purpose of the present invention, methods, apparatus, and computer readable medium for handling an input event directed to a thread within a process operating in a multi-threaded system are disclosed. In one aspect of the present invention, a method is provided in which a process is alerted that an input event effecting one of its active connection threads has been received. A special thread referred to as an input polling thread in the process is enabled and is used, in conjunction with other thread-specific data, to determine which of the threads in the process has an event directed to it. That thread is then triggered to handle the input event.

Brief Summary Text (13):
In one embodiment an execution enabler, such as a light weight process, is assigned to the input polling thread and then run thereby enabling the input polling thread. In yet another embodiment, a list of threads maintained by a process is checked wherein each thread can be identified by a file descriptor and has an associated thread identifier, a thread wait variable, and an error return code. The state of a thread wait variable is changed when an input event directed to a thread associated with the thread wait variable is received.

Brief Summary Text (14):
In another aspect of the invention, a method of invoking a thread in a process when an input event is received and using a reduced number of light weight processes is disclosed. An input event directed at an active connection thread is received and a polling thread is used to determine which active connection thread the input event is directed to. A single light weight process is used by the appropriate active connection thread to handle the input event only such that the light weight process is assigned to the thread only after it is determined that the input event is directed to that thread. In one embodiment a conditional wait thread is run to monitor changes made to the active connection threads receiving an input event to ensure execution of those selected active connection threads.

Brief Summary Text (15):
In another aspect of the invention, a computer system configured to receive input from users where the input is directed to a specific thread contained in a process is disclosed. An input polling thread detects input events directed to active connection threads in the process and routes the event to the selected thread. Only the input polling thread requires a light weight process for detecting the input event thereby reducing the need for light weight processes used by the active connection threads, which would previously be needed for detecting input events. An input wait table associated with the process is structured for monitoring and storing information on the active connection threads where the information

indicates which active connection threads are executing an input event. The input polling thread polls the input wait table to determine which active connection thread an input event is directed to thereby reducing the need for the active connection threads in the process to individually monitor input events.

Brief Summary Text (16):
In one embodiment of the present invention, a conditional wait thread monitors the input wait table to determine which selected active connection thread has had a state change. This ensures that the active connection thread with a state change is assigned a light weight process.

Drawing Description Text (3):
FIG. 1 is a block diagram showing various components of a message access configuration in accordance with one embodiment of the present invention.

Detailed Description Text (3):
A method of reducing the number of light weight processes assigned by an operating system is illustrated in the various drawings. As described above, it is generally desirable for an operating system to have light weight processes available to be assigned to a generic process (e.g. a parent or child process), threads operating within or outside a process, jobs, and other executable entities operating in an operating system. Because the resources available to light weight processes in an operating system is limited, and the amount of resources is proportional to the size of the system, i.e., number of users, amount of data, etc., light weight processes will not have the resources they need to operate if there is significant growth on the system or if the system is not configured to make efficient use of its resources, regardless of the size of the computing environment. Once the pool of light weight processes no longer has resources it can draw from, overall performance of a computer network significantly deteriorates. For example, the operating system may have to perform context switching to accommodate the needs of processes, and users' requests, implemented by threads, must wait for light weight processes thereby slowing response time.

Detailed Description Text (4):
For example, in the described embodiment, a thread represents a client connection. Following this example, the client connection can be to a mail message store on a server in which the user wants to access mail messages. An active connection in this context represents a user session to a message store in a large network. In this environment, a parent process receives user requests to access a mail message store manages several child processes, each containing typically many active connection threads. This configuration of the described embodiment is shown in FIG. 1.

Detailed Description Text (5):
FIG. 1 is a block diagram showing various components of a message access configuration and method in accordance with one embodiment of the present invention. In the described embodiment, the messages stored and accessed are Internet e-mail messages. An Internet Message (IM) access daemon 100 resides on a network mail server, such as an IMAP server which may also contain a message store. An example of an IMAP message store used to store messages and index information which may be used with the present invention is described in co-pending MESSAGE STORE application, the entire specification of which is incorporated herein by reference. A parent process 102 within daemon 100 is responsive to data (typically commands or requests to connect) sent from clients 104. Requests to connect 106 from clients are stored in a queue and are received by the server at a port depending on the protocol in which the request is being made. Once the server responds to a request 106, a connection 108 with a client is established and the client can begin sending data, such as commands for accessing and manipulating mail.

Detailed Description Text (8):
Once a thread is created within a child process, a thread-specific data cell 118 is assigned to that thread. In the described embodiment, this shared memory 116 is created and pre-allocated by the parent process when the server is activated. In other preferred embodiments, the shared memory 116, if needed, can be created by other entities in the operating system. As mentioned, the shared memory is made up of a series of data cells. These cells 118 are identified by a coordinate "i" corresponding to a process and a coordinate "j" corresponding to a thread within that process. Thus, cell (P.sub.i, T.sub.j) is a thread-specific data cell, which also contains a connection number "k," that allows one thread to inform other threads of its actions, such as updating a mailbox or copying messages from a mailbox. The thread-specific data cells 118 of the described embodiment in shared memory 116 allow a thread to inform all other threads under the same parent process of that thread's actions. Thus, the shared memory resides on the server and is pre-allocated and controlled by a parent process once the parent process is invoked.

Detailed Description Text (9):
In prior art systems, each active connection thread would have to have its own light weight process in order to detect any incoming input events directed to it. In a system having a high volume of users, all actively accessing mail and receiving new mail messages, the number of light weight processes can be quickly depleted or reach very low levels. This would result in users on the network not being able to access their mailboxes or not allowing new users to establish connections to the message store. Both these events are highly undesirable given the high performance standards users now expect from e-mail system.

Detailed Description Text (10):
FIG. 2 is an illustration of an input wait table associated with a process in accordance with one embodiment of the present invention. Beginning at the top layer of the method of the described embodiment, the operating system receives an input event. This is generally some type of activity generated by a user or some external source, such as another computer network. In the present invention, the operating system is suited for a distributed computing environment. By way of example, an operating system suitable for use in the present invention can be a Unix-based operating system such as the Solaris.TM. operating system of Sun Microsystems, Inc. or a DOS-based operating system such as the Windows NT.TM. operating system of Microsoft Corp. The input event can be a normal user request, such as checking for new mail or sending a request to print a document, or it can be a less frequent event such as an interrupt signal. Regardless, the input event is normally directed generally to process and specifically to a thread. A method in accordance with one embodiment of the present invention is described fully with regard to FIG. 2. In the preferred embodiments, one table can hold information on more than one process. The table shown in FIG. 2, referred to in the discussion of FIG. 2, is associated generally with a process within the operating system, which contains different types of threads. In addition to being a network-based operating system, the operating system of the present invention is a multi-threaded operating system. A process having threads representing connections to users (or external entities) has an associated input wait table for monitoring and routing input events to an event's corresponding thread or threads. It is possible that in the described embodiment a process can have only one thread or no threads.

Detailed Description Text (11):
In a typical connection, a thread is generally waiting for input or some type of activity to occur the majority of the time the thread is in existence. In the described embodiment, the input wait table for a process stores information on which connections, i.e., threads, are in a wait state and which are executing or processing an input event. In the described embodiment, the operating system assigns to the process a light weight process for maintaining the process's input wait table. The operating system of the described embodiment schedules light weight processes which include assigning them to threads when a thread needs one (e.g.

when a thread makes a call to the operating system or performs an I/O operation). One type of light weight process can be described as an execution enabler that allows a thread to execute. In other preferred embodiments, an operating system may use another type of execution enabler similar to a light weight process, of which there is a limited number available in the system and should be assigned prudently and efficiently. The thread associated with the input wait table invokes a special thread referred to as a polling thread when an input event is received. The polling thread polls or scans the table to determine which connection or connections the input event is directed to and changes the state of the connection from the wait state to a state indicating that it has received input that needs to be processed. It should be noted that the input wait table of the described embodiment may be implemented using other data constructs such as a queue or a linked list. In these embodiments, the polling thread would still have the same function of checking each entry in the data structure for determining which ones have received an input event. Similarly, the receipt of an input event can be manifested in various ways in other preferred embodiments using, by way of example, semaphores, flags, or other types of variables. The polling thread has a single execution enabler assigned to it, which in the described embodiment is a light weight process. As will be discussed in greater detail below, this greatly reduces the need for each connection to have assigned its own light weight process. Once the polling thread has scanned the table and changed the state of the appropriate connections, a conditional wait thread that monitors changes to connection entries in the table is responsible for having any altered connections assigned a light weight process or execution enabler of some sort thereby enabling that now active connection to execute.

Detailed Description Text (12):
Referring now to FIG. 2, an input wait table 200 of the described embodiment includes a file descriptor column 202 for holding file descriptors 204 representing connections, a thread identifier column 206 for holding thread identifiers 208, a conditional wait variable column 210 for holding wait indicators 212, and a error return column 214 for holding error return values 216. In other preferred embodiments, an input wait table may have more columns or fewer columns as appropriate without altering the primary function of the table. As mentioned above, in other preferred embodiments, a process may use a data structure other than a table for storing data on its connections or threads and remain within the scope of the present invention. In the described embodiment, a connection is represented by a row 218 in table 200.

Detailed Description Text (14):
Thread identifier 208 is a unique value that identifies a thread and is different from the thread's file descriptor. The thread identifier 208 is used in connection with posting a thread semaphore or thread conditional wait variable 212. In the described embodiment, the thread conditional wait variable 212 is a flag that indicates whether the thread is in an input wait state or execution state. A polling thread scans the table to determine which thread or threads have input events directed to them and sets variable 212 to "GO." Another thread or light weight process then starts that thread. Thus, wait variable 212 is a thread-specific flag used to indicate the state of a thread. The error return value 216 is used to store any error conditions encountered by the thread. In other preferred embodiments, the input wait table can have additional or fewer columns or fields depending on the particular platform in which the process is running or other system and application requirements. Furthermore, in other preferred embodiments, a process can use another type of data structure to store data regarding threads allowing a polling thread represented by a single light weight process, or similar execution enabler, to manage input events and thread execution.

Detailed Description Text (15):
An input wait table as described above with respect to FIG. 2 is used in a method of receiving an input event and executing the appropriate connection thread while

reducing consumption of light weight processes. FIG. 3 is a flowchart showing a method of executing a connection thread in response to a received input event in accordance with one embodiment of the present invention. At step 302 the operating system detects an input event or some type of activity, for example a user request or a system interrupt. This input event is typically directed to a connection, represented by an active connection thread in a process. The operating system alerts the process containing the active connection thread, where the active connection thread is represented by a file descriptor. In the described embodiment, the system uses the file descriptor associated with the input event to determine which process to alert. As discussed with respect to the file descriptor column 302 of FIG. 3, each thread is represented and identified in the operating system and in the input wait table by a file descriptor 204.

Detailed Description Text (16):
At step 302 the light weight process associated with the process's polling thread is placed in a RUN state. This occurs when a process is alerted that one of its threads has an input event. The polling thread has a light weight process assigned to it but is asleep until its process is alerted, at which point the polling thread is awakened. At step 304 the polling thread begins execution. In the described embodiment, a light weight process allows a connection thread to execute by allowing it to perform necessary functions, such as making system calls or performing input/output operations.

Detailed Description Text (17):
At step 308 the polling thread scans the input wait table to determine which file descriptors, or threads, have an input event. In the described embodiment, it unblocks those file descriptors that have an event by changing the state of the thread's conditional wait variable 212. In the described embodiment this is done by signaling the conditional wait variable to "1" if there is an input event and leaving the thread unaltered if there is no input event for the thread. In the described embodiment, a thread is placed in an input wait state by calling an operating system function called WaitForInput which sets the conditional wait variable 212 to "0" and waits on the conditional wait variable. In other preferred embodiments, the thread or process can call other routines or have other procedures for placing a thread in a wait state by, for example, setting a semaphore or flag.

Detailed Description Text (18):
At step 310 a conditional wait thread in the process that oversees the connection threads to detect which connection threads have had state changes ensures that those threads are assigned light weight processes so they can execute. In the described embodiment, this "conditional wait" light weight process checks to see what the polling thread has done to the connection threads and behaves accordingly. At step 312 of the described embodiment, the thread deletes its entry in the input wait table before it proceeds to process the input event. At step 314 each awakened thread processes its input event or activity, which it can now do since it has been assigned a light weight process. Once the thread has processed the input event, the thread initializes a new entry in the input wait table to indicate its return to the input wait state. In the described embodiment, the conditional wait variable is set to "0" by calling a WaitForInput function and waiting on the conditional wait variable. In other preferred embodiments, the thread can use the same entry it had previously occupied in the input wait table without initializing a new one when it is done processing the input event. Whether the table is static in which rows are reused or dynamic in which new rows are initialized does not effect the underlying functionality and purpose of the input wait table.

Detailed Description Text (19):
Once a new entry for the thread in the input wait table has been initialized, the light weight process allowing execution of the thread is released from that thread. The conditional wait thread mentioned above can then reassign the light weight process to another thread or return it to the pool of light weight processes in the

operating system. By releasing the light weight process from the thread when the thread is done executing, the system realizes benefits of the polling thread. In previous systems, the light weight process would remain with the thread even when the thread was idle or waiting for input. In the present invention, the polling thread using the input wait table and working in conjunction with the "conditional wait" thread, greatly reduces the consumption of light weight processes by connection threads by allowing them to be assigned to threads only when needed.

Detailed Description Text (31):
Although the foregoing invention has been described in some detail for purposes of clarity of understanding, it will be apparent that certain changes and modifications may be practiced within the scope of the appended claims. For instance, the input wait table can contain information on threads in more than one table by having an additional process identifier column. In another example, the input wait table can have more or fewer columns as needed by the system. Furthermore, it should be noted that there are alternative ways of implementing both the process and apparatus of the present invention. Accordingly, the present embodiments are to be considered as illustrative and not restrictive, and the invention is not to be limited to the details given herein, but may be modified within the scope and equivalents of the appended claims.

Other Reference Publication (1):
Crispin, M., "Internet Message Access Protocol," University of Washington, (1996), pp. 1-93.

Other Reference Publication (2):
Lawrence, K.: Writing Multithreaded Graphics Programs, The Developer Connection News, vol. 7, Apr. 1995 (1995-04), pp. 16-17, XP002188186.

Other Reference Publication (3):
Kleiman, S., et al.: Writing Multithreaded Code In Solaris, Intellectual Leverage. San Francisco, Feb. 24-28, 1992, Proceedings of the Computer Society International Conference (COMPCON) Spring, Los Alamitos, IEEE Comp. Soc. Press, US, vol. Conf. 37, (Feb. 24, 1992), pp. 187-192, XP010027136.

CLAIMS:

1. A method of handling an input event to a multi-threaded process that includes a plurality of active connection threads, the method comprising: alerting the multi-threaded process that the input event effecting a selected one of the active connection threads has been received; enabling a polling thread contained in the multi-threaded process; determining the selected one of the active connection threads contained in the multi-threaded process to which the input event is directed using the polling thread operable to scan an input wait table of the active connection threads indicating status corresponding to the active connection threads; and triggering the selected one of the active connection threads to handle the input event.

4. A method as recited in claim 1 wherein determining the active connection thread further includes: checking the list of the active connection threads maintained by the process wherein each thread can be identified by a file descriptor and wherein the list of threads includes for each thread the file descriptor, a thread identifier, a thread wait variable, and an error return code.

5. A method as recited in claim 4 further including changing the state of the thread wait variable when there is an input event directed to a thread associated with the thread wait variable.

6. A method as recited in claim 4 further including enabling a conditional wait thread for monitoring changes to the active connection threads in the list.

7. A method of invoking a thread in a process when the process receives an event directed to the thread, the method comprising: alerting the process, the process having a plurality of threads and associated thread-specific data, that an input event effecting one such thread contained in the process has been received; enabling a polling thread contained in the process; determining which thread contained in the process to which the input event is directed using the polling thread operable to scan an input wait table of the plurality of threads indicating status corresponding to the plurality of threads; and triggering each of the threads in the plurality of threads which has the input event to process that event.

8. A method of handling an input event to a multi-threaded process that includes a plurality of active connection threads, the method comprising: receiving an input event directed at a selected one of the active connection threads; using a polling thread associated with the plurality of the active connection threads to determine which of the plurality of active connection threads the input event is directed to by scanning an input wait table of the active connection threads indicating status corresponding to the active connection threads; and triggering the selected active connection thread to handle the input event, thereby assigning a light weight process to the selected active connection thread to facilitate execution of the selected active connection thread, whereby each active connection thread does not require a dedicated light weight process to facilitate the detection of the input event that is directed to the selected active connection thread, thereby reducing consumption of light weight processes within the multi-threaded process.

9. A method as recited in claim 8 further including running a conditional wait thread to monitor changes made to the plurality of active connection threads having an input event to ensure execution of the selected active connection thread.

10. A computer system having a multi-threaded process capable of receiving an input event from a user directed to a selected one of active connection threads contained in the process, the system comprising: an input polling thread arranged to detect the input event directed to the selected one of the active connection threads in the multi-threaded process and to route the input event to the selected thread whereby only the input polling thread requires a light weight process to detect the input event thereby reducing consumption of light weight processes by the active connection threads used for detecting input events; and an input wait table associated with the process and structured for monitoring and storing information on the active connection threads, such information indicating which one of the active connection threads is executing the input event; whereby the input polling thread polls the input wait table to determine which active connection thread the input event is directed to thereby reducing the need for the active connection threads in the process to individually monitor input events thereby lowering the number of light weight processes running in the computer system.

11. A computer system as recited in claim 10 further comprising a conditional wait thread for monitoring the input wait table to determine which selected one of the active connection threads has had a state change thereby ensuring that the selected one of the active connection threads is assigned to a light weight process.

12. A computer system having a multi-threaded process capable of receiving an input event from a user directed to a selected one of active connection threads contained in the process, the system comprising: a means for detecting the input event directed to the selected one of the active connection threads in the multi-threaded process and to route the input event to the selected thread whereby only the means for detecting requires a light weight process to detect the input event thereby reducing consumption of light weight processes by the active connection threads used for detecting input events; and a means for monitoring and storing information on the active connection threads, such information indicating which one of the

active connection threads is executing the input event; whereby the means for detecting polls the means for monitoring and storing to determine which active connection thread the input event is directed to thereby reducing the need for the active connection threads in the process to individually monitor input events thereby lowering the number of light weight processes running in the computer system.

13. A computer system as recited in claim 12 further comprising a means for monitoring the input wait table to determine which selected one of the active connection threads has had a state change thereby ensuring that the selected one of the active connection threads is assigned to a light weight process.

14. A computer-readable medium containing programming instructions for handling an input event to a multi-threaded process that includes a plurality of active connection threads, the computer-readable medium comprising computer program code devices configured to cause a computer to execute the steps of: alerting the process that the input event effecting a selected one of the active connection threads has been received; enabling a polling thread contained in the process; determining which of the active connection threads contained in the process has the event directed to it using the polling thread operable to scan an input wait table of the active connection threads indicating status corresponding to the active connection threads; and triggering the selected active connection thread to handle the input event.

15. A computer-readable medium as recited in claim 14 further including computer program code devices configured for changing a state of a thread wait variable when there is the input event directed to a thread associated with the thread wait variable.

16. A computer-readable medium as recited in claim 14 further including computer program code devices configured for enabling a conditional wait thread for monitoring changes to active connection threads in the list.

18. A computer-readable medium as recited in claim 14 wherein the computer program code devices configured to cause a computer to determine which of the active connection threads contained in the process has an event directed to it further includes computer program code devices configured to cause a computer to execute the step of checking a list of threads maintained by the process wherein each thread can be identified by a file descriptor and wherein the list of threads includes for each thread the file descriptor, a thread identifier, a thread wait variable, and an error return code.

19. A computer system configured to receive input events from a plurality of users directed to a specific active connection thread contained in a multi-threaded process that includes a plurality of active connection threads, the system comprising: a means for alerting the multi-threaded process that an input event effecting a selected one of the active connection threads has been received; a means for enabling a polling thread contained in the process; a means for determining which of the active connection threads contained in the process has the input event directed to it using the polling thread operable to scan an input wait table of the active connection threads indicating status corresponding to the active connection threads; and a means for triggering the selected active connection thread to handle the input event.

20. A computer-readable medium containing programming instructions for invoking a thread in a process when the process receives an event directed to the thread, the computer-readable medium comprising computer program code devices configured to cause a computer to execute the steps of: alerting the process, the process having a plurality of threads and associated thread-specific data, that an input event effecting a thread contained in the process has been received; enabling a polling

thread contained in the process; determining which of the threads contained in the process has the input event directed to it using the polling thread operable to scan an input wait table of the plurality of the threads indicating status corresponding to the plurality of the threads; and triggering each of the threads in the plurality of threads which has the input event to process that event.

21. A computer-readable medium containing programming instructions for handling an input event to a multi-threaded process that includes a plurality of active connection threads, the computer-readable medium comprising computer program code devices configured to cause a computer to execute the steps of: receiving an input event directed at a selected one of the active connection threads; using a polling thread associated with the plurality of the active connections threads to determine which one of the plurality of active connection threads the input event is directed to by scanning an input wait table of the active connection threads indicating status corresponding to the active connection threads; and triggering the selected active connection thread to handle the input event, thereby assigning a light weight process to the selected active connection thread to facilitate execution of the selected active connection thread, whereby each active connection thread does not require a dedicated light weight process to facilitate the detection of the input event that is directed to the selected one of the active connection threads, thereby reducing consumption of light weight processes within the multi-threaded process.

22. A computer-readable medium as recited in claim 21 further including computer program code devices configured for running a conditional wait thread to monitor changes made to the plurality of active connection threads having an input event to ensure execution of the selected active connection thread.

23. A computer system configured to receive input events from a plurality of users directed to a specific active connection thread contained in a process, the system comprising: a means for receiving an input event directed at a selected one of the active connection threads; a means for using a polling thread associated with the plurality of the active connections threads to determine which one of the plurality of active connection threads the input event is directed to by scanning an input wait table of the active connection threads indicating status corresponding to the active connection threads; and a means for triggering the selected active connection thread to handle the input event, thereby assigning a light weight process to the selected active connection thread to facilitate execution of the selected active connection thread, whereby each active connection thread does not require a dedicated light weight process to facilitate the detection of the input event that is directed to the selected one of the active connection threads, thereby reducing consumption of light weight processes within the multi-threaded process.

24. A computer system as recited in claim 23 further comprising a means for running a conditional wait thread to monitor changes made to the plurality of active connection threads having an input event to ensure execution of the selected active connection thread.

25. A computer system configured to receive input events from a plurality of users directed to a specific active connection thread contained in a process, the system comprising: a means for alerting the process, the process having a plurality of threads and associated thread-specific data, that an input event effecting a thread contained in the process has been received; a means for enabling a polling thread contained in the process; a means for determining which of the threads contained in the process has the input event directed to it using the polling thread operable to scan an input wait table of the plurality of the threads indicating status corresponding to the plurality of the threads; and a means for triggering each of the threads in the plurality of threads which has the input event to process that event.

Hit List

Clear | Generate Collection | Print | Fwd Refs | Bkwd Refs | Generate OACS

Search Results - Record(s) 1 through 1 of 1 returned.

☐ 1. Document ID: US 6263370 B1

L24: Entry 1 of 1 File: USPT Jul 17, 2001

US-PAT-NO: 6263370
DOCUMENT-IDENTIFIER: US 6263370 B1

TITLE: TCP/IP-based client interface to network information distribution system servers

DATE-ISSUED: July 17, 2001

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Kirchner; Michael C.	Cedar Rapids	IA		
Scharf; David C.	Dallas	TX		
Herder; Thomas J.	Cedar Rapids	IA		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
MCI Communications Corporation	Washington	DC			02

APPL-NO: 08/ 923622 [PALM]
DATE FILED: September 4, 1997

INT-CL: [07] G06 F 15/16

US-CL-ISSUED: 709/230; 709/219, 709/229
US-CL-CURRENT: 709/230; 709/219, 709/229

FIELD-OF-SEARCH: 709/229, 709/228, 709/227, 709/226, 709/237, 709/236, 709/219, 709/230, 370/465, 370/469, 370/471

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
4922486	May 1990	Lidinsky et al.	370/427
5345396	September 1994	Yamaguchi	709/237
5436627	July 1995	Motoyama et al.	341/67

h e b b g e e e f e h ef b e

5438650	August 1995	Motoyama et al.	707/524
5638066	June 1997	Horiuchi et al.	370/476
5706437	January 1998	Kirchner et al.	709/227
5764915	June 1998	Heimsoth et al.	709/227
5875178	February 1999	Rahuel et al.	370/471

OTHER PUBLICATIONS

Cheah, R. S.-S. et al.; "Implementation of OSI application layer using ISO Development Environment"; IEEE International Conference on Information Engineering; ISBN 0-7803-1445-X; pp. 28-33, v1, Sep. 1993.*
Cheah, R. S.-S. et al.; "Implementing Manufacturing Message Specification services and protocol using ISO Development Environment"; IEEE Region 10 Conference on Computer, Communication, Control, and Power Engineering; ISBN 0-7803-1233-3; pp. 553-556, v1, Oct. 1993.*
Steven, Richard W.; "TCT/IP Illustrated, vol. 1, The Protocols"; Addison-Wesley Publishing; ISBN 0-201-63346-9; Chapters 1, 17, 25, 28, Dec. 1994.*
Stark, Thom; "Internet Mail is a MIME field"; LAN Times; McGraw-Hill Inc.; Feb. 5, 1996 v13, n3, p110(1), Feb. 1996.

ART-UNIT: 212

PRIMARY-EXAMINER: Harrell; Robert B.

ASSISTANT-EXAMINER: Thompson; Marc D.

ABSTRACT:

A TCP/IP-based client-server interface for sending service requests from a client computer to a Network Information Distribution Services server. The TCP/IP-based client-server interface is an easily implemented and economical alternative to the proprietary UDP/IP-based client-server interface developed for communication between client computers and Network Information Distribution Services servers. A service request is sent to a specific TCP/IP logical port associated with a particular type of service. A process associated with the specific TCP/IP logical port formats the request in the same manner as requests received through the proprietary UDP/IP-based client-server interface are formatted, and then directs the formatted requests to the Network Information Distribution Services server process that executes the requests.

10 Claims, 14 Drawing figures

Term	Documents
(22 AND 23).USPT.	1
(L23 AND L22).USPT.	1

h e b b g e e e f e h ef b e

Display Format: FRO Change Format

Previous Page Next Page Go to Doc#

h e b b g e e e f e h ef b e

L44: Entry 2 of 6 File: USPT Apr 24, 2001

DOCUMENT-IDENTIFIER: US 6223207 B1
TITLE: Input/output completion port queue data structures and methods for using same

Abstract Text (1):
A technique for performing multiple simultaneous asynchronous input/output operations in a computer operating system. An input/output completion port object is created and associated with a file descriptor. When I/O services are requested on the file descriptor, completion is indicated by a message queued to the I/O completion port. A process requesting I/O services is not notified of completion of the I/O services, but instead checks the I/O completion port's queue to determine the status of its I/O requests. The I/O completion port manages multiple threads and their concurrency.

Brief Summary Text (18):
The foregoing problems are exacerbated in multi-processing operating systems, in which several different applications are alternately executing and "sleeping" to effect apparent simultaneous execution. In this environment there is the further difficulty of correlating hardware interrupts to the applications to which they correspond--applications which may be "asleep" when the I/O completion interrupt corresponding thereto is returned.

Brief Summary Text (21):
While an improvement over earlier approaches, select has numerous problems of its own. One is that the temporal information provided to select by the order in which it receives the interrupts is lost; there is no way for the application programs to learn which I/O operations have been completed the longest. This is of particular concern in multi-tasking systems, in which ensuring "fairness" between concurrent processes requires knowledge of which I/O results have been waiting the longest for further processing.

Brief Summary Text (23):
A still further drawback of select( ) is its inability to deal with multi-threaded processes. Such processes are widely used in many newer operating systems, such as Microsoft Windows NT.

Brief Summary Text (24):
A multi-threaded process has two more threads for process execution within a single process. Each thread shares the same address space, descriptors and other resources within the process, but has its own program counter for execution. To achieve concurrency using threads, an application program creates two or more threads within a process to execute different parts of the program within the same process. A multi-threaded process may be used for i/O operations on several descriptors, and can be used for I/O operations that happen on the same descriptor. A similar approach in the prior art splits a single-threaded process into multiple processes for execution. However, since processes require significantly more operating system overhead than threads, the multi-threaded process approach is preferred.

Brief Summary Text (25):
Select( ) cannot handle multiple threads or the concurrency that is required by

simultaneous asynchronous I/O operations on the same descriptor, or different descriptors.

Brief Summary Text (26):
Another method used in the prior art to provide information about the occurrence of an I/O event is to send a messages to the corresponding file handle. Message queues are used to collect and pass message information in both threaded process environments, and non-threaded process environments. For example Windows 3.x uses message queues to route messages to windows applications. Windows 3.x maintains a single system message queue and any number of thread message queues, one for each thread. Whenever a user moves the mouse, clicks mouse buttons, or types at the keyboard (I/O events), the device driver for the mouse or keyboard converts the input into messages and places them in the system wide message queue. A Windows message handler removes the messages, one at a time, from the system message queue, examines them to determine the destination thread, and then posts them to message queue of the thread. A thread's message queue receives all mouse and keyboard messages from the system wide queue and directs the Windows kernel to send them to the appropriate Window's application associated with the thread for processing.

Brief Summary Text (27):
One disadvantage to sending all messages (including I/O messages) to a single system wide queue is that Windows must sort through and process a large number of different types of messages. This is expensive in terms of processing time. A queue per thread also is expensive in terms of system resources and system overhead. If a large number of threads are used, then a large number of thread queues have to be created, managed, and then deleted. Windows must also maintain a table of addresses for all the thread queues. Another disadvantage to this approach is that every message is "handled" at least three times (e.g. once by Windows to take it out of the single system queue, once by the thread to remove it from the thread queue, and once by the Windows application that must process the message). Handling every message several times causes significant operating system overhead and delaying the processing of individual applications that may be waiting for messages.

Brief Summary Text (28):
In accordance with a preferred embodiment of the present invention, the foregoing and other disadvantages of the prior art are overcome with an I/O completion port object. The I/O completion port object is an I/O object with a queue that provides a single synchronization point and controllable concurrency for multiple simultaneous asynchronous I/O operations. These multiple simultaneous asynchronous I/O operations could be the result of I/O requests from a single computer system, or a network of computer systems. For example, an I/O completion port can be used to synchronize hundreds of network I/O operations.

Brief Summary Text (29):
An I/O completion port is created and associated with a file descriptor. Any number of file descriptors can then be associated with a single I/O completion port. If a process creates a number of threads to complete an operation on a single or on multiple descriptors, then all the threads are also associated with an I/O completion port. As a result, the I/O completion port provides concurrency control for these multiple threads.

Brief Summary Text (30):
Once a file descriptor is associated with an I/O completion port, the completion of any subsequent I/O request on that descriptor causes an I/O completion port packet to be queued to the I/O completion port. The I/O completion port completion packet contains information about the I/O request (e.g. success, failure, amount of information transferred, etc.). Instead of having the I/O system contact the application which made the I/O request, the application checks the I/O completion port's queue to determine if the I/O request has been completed. The I/O completion port completion packet is then used to determine the state of the completed I/O and

initiate any subsequent action.

Brief Summary Text (31):
The I/O completion port allows tracking of I/O operations not only per descriptor, but also per I/O operation. If multiple threads are created to complete the per I/O operation, then the concurrency of the threads is handled by the I/O completion port. For example, if a large read operation is split into several smaller read operations (several threads) of a certain block size, the I/O completion port can track the completion of the reads. If read number two finishes before read number one, the I/O completion port can be used by the application to determine which read was which.

Drawing Description Text (6):
FIG. 3B is a depiction of a kernel queue object.

Detailed Description Text (12):
The bottommost portions of the executive are called the "kernel" 42 and the "hardware abstraction layer" ("HAL") 44. The kernel 42 performs low-level operating system functions, such as thread scheduling, interrupt and exception dispatching, and multiprocessor synchronization. The hardware abstraction layer (HAL) 44 is a layer of code that isolates the kernel and the rest of the executive from platform-specific hardware differences. The HAL thus hides hardware-dependent details such as I/O interfaces, interrupt controllers, and multiprocessor communication mechanisms. Rather than access hardware directly, the components of the executive maintain portability by calling the HAL routine when platform-specific information is needed.

Detailed Description Text (15):
The executive 40 is a series of components, each of which implements two sets of functions: system services 54, which subsystems and other executive components can call, and a set of internal routines, which are available only to components within the executive. System services include (a) the object manager 40a, which is responsible for creating, managing and deleting objects (objects are abstract data structures used to represent operating system resources); (b) the process manager 40b, which is responsible for creating/terminating processes and threads, and for suspending/resuming execution of threads; and (c) the I/O manager 40f, which is responsible for implementing device-independent input/output (I/O) facilities as well as device-dependent I/O facilities.

Detailed Description Text (16):
The client 52 requests a service by sending a message to a subsystem 48, as represented by the solid arrow between the depicted Win32 client 52 and the Win32 subsystem 48. The message passes through system services 54 and the executive 40, which delivers the message to the subsystem. After the subsystem 48 performs the operation, the results are passed to the client 52 in another message, as represented by the dotted arrow between the Win32 subsystem and the Win32 client.

Detailed Description Text (17):
In Windows NT, shareable resources, such as files, memory, processes and threads, are implemented as "objects" and are accessed by using "object services." As is well known in the art, an "object" is a data structure whose physical format is hidden behind a type definition. Data structures, also referred to as records or formats, are organization schemes applied to data so that it can be interpreted and so that specific operations can be performed on that data. Such data structures impose a physical organization on the collection of data stored within computer memory and represent specific electrical or magnetic elements.

Detailed Description Text (19):
The Windows NT operating system allows users to execute more than one program at a time by organizing the many tasks that it must perform into "processes" 46-52. The

operating system allocates a portion of the computer's resources to each process and ensures that each process's program is dispatched for execution at the appropriate time and in the appropriate order. In the preferred embodiment, this function is implemented by the process manager 40c.

Detailed Description Text (24):
at least one "thread of execution."

Detailed Description Text (25):
A "thread" is the entity within a process that the kernel schedules for execution. As is well known in the art, each thread has an associated "context" which is the volatile data associated with the execution of the thread. A thread's context includes the contents of system registers and the virtual address belonging to the thread's process. Thus, the actual data comprising a thread's context varies as it executes.

Detailed Description Text (26):
Periodically, a thread may stop executing while, for example, a slow I/O device completes a data transfer or while another thread is using a resource it needs. Because it would be inefficient to have the processor remain idle while the thread is waiting, a multi-tasking operating system will switch the processor's execution from one thread to another in order to take advantage of processor cycles that otherwise would be wasted. This procedure is referred to as "context switching." When the I/O device completes its data transfer or when a resource needed by the thread becomes available, the operating system will eventually perform another context switch back to the original thread. Because of the speed of the processor, both of the threads appear to the user to execute at the same time.

Detailed Description Text (28):
The Windows NT kernel is responsible for allocating the available threads and completing context switches in an efficient manner. To do this, Windows NT used a priority-based round-robin algorithm. This algorithm ensures no process or thread will be "starved" out of execution time (i.e. every process or thread will get a chance to execute periodically for a certain amount of time).

Detailed Description Text (29):
The Windows NT kernel supports 31 different priorities. There is a queue for each of the 31 priorities that contains all the ready threads at that priority. When a CPU becomes available, the kernel finds the highest priority queue with ready threads on it, removes the thread at the head of the queue, and runs it. This is the "context switch."

Detailed Description Text (30):
The most common reason for a context switch is when a running thread needs to wait for an event to complete. For example, the ReadFile( ) function call blocks the running thread until the specified read event completes. When a running thread needs to wait, the kernel picks the highest-priority ready thread and switches it to the running state. This ensures that the highest priority runnable threads are always running. This also prevents CPU-bound threads from monopolizing the processor, since the kernel imposes a time limit on each thread called a thread quantum. When a thread has been running for one quantum, the kernel preempts it and puts it at the end of the ready queue for its priority. On a uni-processor computer system, only one thread can be in the running state. A multi-processor computer system allows one thread per processor to be in the running state.

Detailed Description Text (31):
When the event a blocked thread is waiting for completes, the blocked threat is put on the end of the ready queue for its priority by the kernel. At some point, the thread which was blocked will be picked and executed by the kernel, completing the required transaction.

Detailed Description Text (33):
The preferred embodiment of the present invention makes use of operating system data structures called "queue objects" which control the number of threads that are actively processing incoming requests to the operating system (otherwise known as the concurrency level). Queue objects keep track of how many threads are currently active, and ensure that the number of active threads is at or near a predetermined target level of concurrency. By ensuring that new threads are not added to the pool of active threads if the system is operating at or above the target level of concurrency, the preferred embodiment minimizes the number of superfluous context switches that the kernel must perform.

Detailed Description Text (34):
Queue objects, which are created by the operating system as needed and stored within the memory device 16, are "kernel objects," i.e., they are not visible to user-mode code. In other words, queue objects are used only within the OS kernel 34 (FIG. 2) and may therefore also be referred to-as "kernel queue objects."

Detailed Description Text (35):
An exemplary kernel queue object 58 (sometimes referred to simply as a queue) is shown in FIG. 3B and is formatted as follows:

Detailed Description Text (36):
Size 58a: the size of the queue object. In the preferred embodiment, the size of the queue object is fixed at 32 bytes. However, other sizes can, of course, be used.

Detailed Description Text (37):
Type 58b: the name of the queue object of this type.

Detailed Description Text (38):
State 58c: the number of "work entries" in the entry list, as defined below. A "work entry" is any request that is placed in the queue object to be distributed to a thread for execution.

Detailed Description Text (39):
Wait List 58d: a list of threads that are waiting for a set of conditions to be satisfied so they can be removed from the wait list and placed in the ready queue. In the preferred embodiment, threads are waiting either for a work entry to be inserted into the entry list, as defined below, and/or for the "concurrency level" to drop below the target maximum number of threads. The "concurrency level" is defined as the current number of active threads.

Detailed Description Text (40):
Entry List 58e: a list of work entries to be distributed to the pool of threads for execution.

Detailed Description Text (41):
Target Maximum Number of Threads 58f: the target number of threads that should be active processing queue object entries concurrently. In general, this value is set to the desired level of concurrency with respect to the queue object and is therefore also referred to as the "target concurrency level." In the preferred embodiment, the target maximum number of threads has a default value set to the number of processing units in the system, but is not limited to this value.

Detailed Description Text (42):
Current Number of Active Threads 58g: the number of threads that are currently active, i.e., in the ready state, the standby state, the running state, or the transition state. This number is also referred to as the "current concurrency level" or simply the "concurrency level."

Detailed Description Text (43):
The preferred embodiment provides a kernel queue object called an "I/O completion port object" (FIG. 4) that reports I/O completion status of an open file or device. The I/O completion port object is created by a user mode 36 component (e.g. a subsystem or an application program) by calling the executive 40. In response to this call, the object manager 40a within the executive creates an I/O completion port object, an optional name for the object, and an optional access control list, and returns a handle that the user mode component can refer to when referencing the I/O completion port object.

Detailed Description Text (44):
In operation, when an I/O completion port object is referred an I/O request, the I/O completion information is placed in the I/O completion port object entry list. The user mode subsystem or application program then has threads that wait on the I/O completion port object.

Detailed Description Text (45):
Referring to FIG. 4, the I/O completion port object 60 includes an object header 62a with an embedded kernel queue object 62b. The object header and attributes of the I/O completion port object follow the standard conventions for kernel objects in the Windows NT operating system.

Detailed Description Text (46):
As noted in the background discussion, one of the problems with asynchronous I/O arises in the context of multi-threaded operating systems. Completion of an asynchronous I/O request must be reported to the thread that issued the I/O request.

Detailed Description Text (47):
The I/O completion port object provides a solution to this problem. Referring to FIG. 5, completion port object 62 and overlapped I/O 64 are used to support multiple clients on a single thread. When an I/O request is completed, the completion information is routed by the Windows NT I/O subsystem 66 to a completed I/O queue 68 within the I/O object completion port object 62, rather than directly to the thread that requested the I/O operation. A small set of threads 70 are provided that wait on the queue object to which all I/O completion information is sent. Thus, I/O completion is not tied to the thread that issued the I/O request. When a queue entry 68 (i.e., the I/O completion) appears in the queue object, and there is a thread 70 waiting to process that entry, and the maximum target level of concurrency is not exceeded, then the queue entry is given to the waiting thread. Thus, the I/O completion port object can service a large number of incoming requests with a much smaller pool of threads than in current synchronization methods.

Detailed Description Text (48):
Since the existing Windows NT service WaitForMultipleObjects can be used to support multiple clients on a single thread, the I/O completion port object was created to provide additional functionality. One important new feature provided by the I/O completion port is controllable concurrency. An I/O completion port's concurrency value is specified when it is created. This value limits the number of runnable threads associated with the port. When a thread waits on a completion port, the executive 40 associates the thread with the port it is waiting on. The executive is then responsible for trying to prevent the total number of runnable threads associated with an I/O completion on that port from exceeding the port's concurrency value (i.e. throttling). It does this by blocking threads 70 waiting on an I/O completion port until the total number of runnable threads associated with the port drops below its concurrency value.

Detailed Description Text (49):

As a result, when an application thread 72 calls the I/O completion port (via the GetQueuedCompletionStatus service request 74 explained below), the thread only returns when there is completed I/O available, and the number of runnable threads associated with the I/O completion port is less than the port's concurrency. The executive 40 dynamically tracks the completion port's runnable threads. When one of these threads blocks, the executive checks to see if it can awaken a blocked thread waiting on the I/O completion port to take its place. This throttling effectively prevents the system from becoming swamped with too many runnable threads. Since there is one central synchronization point for all the I/O, a small pool of worker threads can service many clients.

Detailed Description Text (50):
Unlike the rest of the Windows NT synchronization objects, threads that block on waits to an I/O completion port are unblocked in a last-in-first-out (LIFO) order 70. Since it does not matter which thread services an I/O completion, the most recently active thread is woken up. Threads at the bottom of the stack have been waiting for a long time and will usually continue to wait, allowing the system to swap most of their memory resources out to disk. Threads near the top of the stack are much more likely to have run recently, so their memory resources will not be swapped to disk or flushed from the processor's cache. The net result is that the number of threads waiting on the I/O completion port is not very important. If more threads than are needed block on the port, the unused threads simply remain blocked. The system will be able to reclaim most of their resources, but the threads will remain available if there are enough outstanding transactions to require their use. A dozen threads can easily service a large set of clients, although this will vary depending on how often each transaction needs to wait. Note that the LIFO policy only applies to threads that block on the I/O completion port. The I/O completion port delivers completed I/O in first-in-first-out (FIFO) order 68.

Detailed Description Text (51):
Tuning the I/O completion port's concurrency is a little more complicated. The best value to pick is usually one thread per CPU. This is the default if zero is specified at the creation of the I/O completion port. There are a few cases where a larger concurrency is desirable. For example, if the I/O transaction requires a lengthy computation that will rarely block, a larger concurrency value will allow more threads to run. The executive 40 will preemptively timeslice among the running threads, so each transaction will take longer to complete. However, more transactions will be processing at the same time, rather than sitting in the I/O completion port's queue, waiting for a running thread to complete. Depending on the application, this may provide quicker response to clients. Simultaneously processing more transactions allows applications to have more concurrent outstanding I/O, resulting in higher utilization of the available I/O throughput.

Detailed Description Text (52):
As discussed above, queue objects are "kernel objects," i.e., they are created and used only within the kernel, and only components that run in kernel mode 50 can access and use queue objects directly. Therefore, user mode subsystems and application programs cannot use queue objects like the I/O completion port object unless there is a "kernel object" that exports the capabilities of queue objects to user mode 36.

Detailed Description Text (54):
The CreateIoCompletionPort( ) function call provides a way to create an I/O completion port handle, and attach application file handles to the created I/O completion port. CreateIoCompletionPort( ) takes 4 arguments: FileHandle, CompletionPort, CompletionKey, and NumberofConcurrentThreads. FileHandle is a file handle defining a client that will use the I/O completion port, CompletionPort is handle defining the I/O completion port, CompletionKey is a double word variable that contains an application defined per-file handle context. When an I/O operation

completes, this value is returned as part of the return values from GetQueuedCompletionStatus( ). NumberofConcurrentThreads is a double word defining the number of threads to execute concurrently.

Detailed Description Text (55):
The GetQueuedCompletionStatus( ) function call provides a way for an application to get I/O completion packets from the I/O completion port. GetQueuedCompletionStatus ( ) takes 5 arguments: CompletionPort, &nbytes, &WorkIndex, &lpo, WaitTime. CompletionPort is the handle identifying the I/O completion port returned from CreateIoCompletionPort( ). The argument &nbytes is the address of a variable which will be filled with the number of bytes read and/or written for the I/O request. The argument &WorkIndex is the address of a double word which will be filled with the file handle context that was established as the value CompletionKey in the CreateIoCompletionPort( ) call. The argument &lpo is the address of the Windows NT lpOverlapped structure which contains fields for determining the number of I/O errors that may have occurred with the I/O service request, and the context of the I/O. The argument WaitTime is a timeout value for how long the call to GetQueuedCompletionStatus( ) should wait before returning-if there are no I/O completion packets returned to the I/O completion port queue for the application. The I/O port completion packet (explained below) is obtained from a call to GetQueuedCompletionStatus( ).

Detailed Description Text (56):
The PostQueuedCompletionStatus( ) function call allows an application to queue I/O completion packets directly to the I/O completion port. The PostQueuedCompletionStatus( ) takes four arguments: CompletionPort, NumberofBytesTransferred, CompletionKey, and lpOvelapped. CompletionPort supplies a handle to the completion port that the caller wants to post a completion packet to. NumberofBytesTransferred is a double word which supplies the value that is to be returned in the lpNumberofBytesTransfered parameter of the GetQueuedCompletionstatus( ) application program interface (API). CompletionKey is a double word that supplies the value that is to be returned through the lpCompletionKey parameter of the GetQueuedCompletionStatus( ) API. The lpOverlapped argument is an LPOVERLAPPED structure that supplies the value that is to be returned through the lpOverlapped parameter of the GetQueuedCompletionStatus( ) API. The use of PostQueuedCompletionStatus( ) function call will be explained below.

Detailed Description Text (60):
Thereafter, when an asynchronous I/O request initiated on a file handle associated with the I/O completion port object completes, an I/O completion packet is queued to the I/O completion port 62. The contents of the I/O completion packet are explained below.

Detailed Description Text (61):
When an asynchronous I/O completion is received by the Windows NT subsystem 66 (FIG. 5), if the current number of runnable threads for the I/O completion port is less than the port's concurrency, the I/O completion port object will be signaled. If there are threads waiting on the port, the executive 40 will wake up the last thread 70 (waits on I/O completion ports are satisfied in LIFO order) and hand it the I/O completion packet. If there is no thread currently waiting on the port, the packet will be handed to the next thread that calls GetQueuedCompletionStatus.

Detailed Description Text (62):
The most efficient scenario occurs when there are I/O completion packets waiting in the queue 68, but no waits can be satisfied because the port has reached its concurrency limit. In this case, when a running thread completes a transaction, it calls GetQueuedCompletionStatus to pick up its next transaction and immediately picks up the queued I/O packet. The running thread never blocks, the blocked threads never run, and no context switches occur. This demonstrates one of the most

interesting properties of I/O completion ports: the heavier the load on the system, the more efficient they are. In the ideal case, the worker threads never block, and I/O completes to the queue at the same rate that threads remove it. There is always work on the queue, but no context switches ever need occur. After a thread completes one transaction, it simply picks the next one off the completion port and keeps going.

Detailed Description Text (63):
The queuing of the I/O completion port packets is performed by the Windows NT I/O manager 40f and is completely transparent to the application that initiated the I/O operation. The kernel queue object is used by the I/O manager for the queuing of completed I/O packets. This allows an I/O completion port object to benefit from all the features of a kernel queue object, particularly the controllable concurrency level.

Detailed Description Text (64):
To receive notification of completed I/O, an application 72 tries to remove I/O completion packets from the I/O completion ports queue with the GetQueuedCompletionStatus( ) 74 service. The following C/C++ language code fragment shows how an application may obtain I/O completion messages from the I/O completion part.

Detailed Description Text (67):
An application may also queue I/O completion packets directly to the I/O completion port with the PostQueuedCompletionStatus( ) method. The application must specify the contents of the entire I/O completion packet, and the resulting packet is queued directly to the I/O completion port with no interaction with the I/O manager 40f. This approach may be used in special circumstances when an application wants to avoid using the normal I/O completion port method.

Detailed Description Text (68):
A method of selectively initiating I/O that will not cause an I/O completion packet to be queued on its completion is also provided. Under Windows NT, this is initiated through the same methods as other I/O. A bit in the hEvent field of the lpOverlapped structure is used to indicate to the I/O manager that the completed I/O resulting from this particular operation should not be queued to the I/O completion port, but should complete through the normal mechanism.

Detailed Description Text (69):
Having illustrated and described the principles of the present invention in a preferred embodiment, it should be apparent to those skilled in the art that the embodiment can be modified in arrangement and detail without departing from such principles. For example, while the invention has been described in the context of an object-oriented operating system, it will be recognized that the invention finds applicability in other types of operating systems as well. Similarly, while the invention has been illustrated with reference to a queue as the operative data structure, those skilled in the art will recognize that other data structures can be used to similar effect.

Detailed Description Text (70):
Likewise, it will be recognized that other embodiments of the present invention can incorporate the principles thereof in conjunction with interrupt notification techniques. For example, the assembly and transmission to a queue of an I/O notification packet can be accomplished by a process that responds to I/O interrupts.

Detailed Description Text (72):
More fundamentally, while the invention has been illustrated with reference to queuing completion messages from I/O operations, the invention is not so limited. Instead, it finds applicability in any context in which a service is asynchronously

requested from the operating system by an application program.

CLAIMS:

1. A computer-readable medium storing instructions for an input/output completion port queue data structure for use in a computer system with one or more processors, an operating system and a memory, the operating system being a multi-tasking operating system that supports a plurality of processes with a group of executable threads, a plurality of worker threads out of the group of executable threads requesting to service input/output completions on the input/output completion port queue data structure in the memory of the computer system, wherein a worker thread that requests to service an input/output completion blocks if the input/output completion port queue data structure fails a readiness criterion, the input/output completion port queue data structure comprising:

a first queue for storing input/output completion messages, the first queue used in a first-in-first-out manner so as to distribute input/output completion messages to the worker threads in the order the first queue receives the input/output completion messages, wherein one readiness criterion is whether the first queue currently stores any input/output completion messages; and

a second queue for ordering blocked worker threads, the second queue used in a last-in-first-out manner so as to unblock blocked worker threads in the reverse of the order that the worker threads block, whereby the worker threads efficiently utilize system resources.

3. The I/O completion port queue data structure of claim 1 in which a small number of worker threads service a majority of the input/output completions.

4. A method of using the input/output completion port queue data structure of claim 1, the method comprising:

requesting by a worker thread to service an input/output completion on the data structure; and

determining whether the first queue currently stores any input/output completion messages,

and if so, servicing by the worker thread an input/output completion,

and if not, blocking the worker thread.

5. The method of claim 4 which further includes:

based upon the ordering of the second queue, allowing the operating system to swap operating system resources occupied by such blocked worker thread to secondary storage, thereby increasing the number of operating system resources available to other threads.

6. The method of claim 4 further comprising:

for a first input/output completion message, unblocking the blocked worker thread; and

servicing by the worker thread an input/output completion corresponding to the first input/output completion message.

7. The method of claim 4 further comprising:

at a later time, requesting by a second worker thread to service an input/output

completion on the data structure;

blocking the second worker thread;

for a first input/output completion message, unblocking the blocked second worker thread while the first worker thread remains blocked; and

servicing by the second worker thread an input/output completion corresponding to the first input/output completion message.

8. The method of claim 7 further comprising:

requesting again by the second worker thread to service an input/output completion on the data structure; and

servicing by the second worker thread an input/output completion corresponding to a second input/output completion message that the first queue receives after receiving the first input/output completion message.

9. The method of claim 7 further comprising:

for a second input/output completion message that the first queue receives after receiving the first input/output completion message, unblocking the first worker thread; and

servicing by the first worker thread an input/output completion corresponding to the second input/output completion message.

10. The method of claim 4 further comprising:

providing an executive process in the operating system that tracks the number of active worker threads using a multi-tasking concurrency control.

11. A method of using the input/output completion port queue data structure of claim 1, the method comprising:

requesting to service input/output completions on the data structure by first and second worker threads;

blocking each of the requesting worker threads until the first queue stores an input/output completion message, wherein the second worker thread blocks after the first worker thread;

for a first input/output completion message,

unblocking the second worker thread while the first worker thread remains blocked; and

servicing by the second worker thread an input/output completion for the first input/output completion message.

12. The method of claim 11 wherein a requesting worker thread specifies a time as a timeout value, and wherein if the requesting worker thread fails to service an input/output completion within the specified time, the requesting worker thread unblocks.

13. The method of claim 11 further comprising:

requesting again to service an input/output completion on the data structure by the second worker thread;

servicing by the second worker thread an input/output completion for a second input/output completion message that the first queue receives after receiving the first input/output completion message.

14. The method of claim 11 further comprising:

for a second input/output completion message that the first queue receives after receiving the first input/output completion message, unblocking the first worker thread; and

servicing by the first worker thread an input/output completion corresponding to the second input/output completion message.

15. A method of using the input/output completion port queue data structure of claim 1, the method comprising:

requesting to service an input/output completion on the data structure by a worker thread; and

determining if the first queue currently stores any input/output completion messages, and if so,

servicing by the requesting worker thread an input/output completion and, if not,

blocking the requesting worker thread, whereby system resources for the blocked requesting worker thread that are not immediately needed to process input/output completion messages are eventually released to promote efficient utilization of system resources.

16. The method of claim 15 wherein the requesting worker thread specifies a time as a timeout value, and wherein if the requesting worker thread fails to service an input/output completion within the specified time, the requesting worker thread unblocks.

17. The method of claim 15 wherein servicing comprises processing an input/output completion message from the first queue.

18. The object-oriented input/output completion port queue data structure of claim 1 wherein an input/output completion message comprises:

a first data field containing data representing a quantity of information handled in an input or output operation;

a second data field containing data representing error indications for the input or output operation;

a third data field containing data representing a context for the input or output operation; and

a fourth data field containing data representing a context for a thread that requested the input or output operation, whereby the input/output completion message provides completion information for any thread that requests an input/output operation.

19. The input/output completion port queue data structure of claim 1 wherein the multi-tasking operating system comprises an executive layer and a kernel layer, wherein the input/output completion port queue data structure resides in the kernel layer, and wherein the plurality of processes with a group of executable threads request services from the input/output completion port queue data structure through

the executive layer.

22. In a computer system that includes one or more processors, an operating system, and plural execution threads, the operating system scheduling and switching runnable execution threads to effect multi-tasking, the system further including a memory device coupled to the one or more processors and accessible by said operating system, a method comprising:

requesting first and second operations by one or more of plural runnable execution threads of an application program;

for a completed operation, transmitting a completion message from the operating system to a queue stored in said memory, wherein the queue stores a first completion message for the first operation after the first operation completes, wherein the queue stores a second completion message for the second operation after the second operation completes, and wherein the ordering of the first and second completion messages in the queue reflects the order that the first and second operations complete;

checking the queue for completion messages by one or more of plural worker threads;

with the one or more worker threads, processing the first and second completion messages in the order they were enqueued, whereby processing of completions for the first and second operations is based on the order the requested operations complete.

23. In a computer system that includes one or more processors, an operating system and one or more application programs running on said computer system, the operating system being a multi-tasking operating system that supports execution of plural processes with one or more runnable threads, wherein one or more of the runnable threads is used to execute an individual application program, and wherein one or more worker threads out of the group of runnable threads request to service input/output completions on a queue object, the system further including a memory device coupled to the one or more processors and accessible by said operating system, a method for performing asynchronous operations comprising:

providing a queue object in said memory, said queue object including at least two queues, said queue object having one or more worker threads associated therewith for processing completion messages;

requesting an asynchronous operation by a runnable thread of an application program;

on completion of the requested asynchronous operation, adding a completion message to a first of said queues, wherein the first queue operates in a first-in-first-out manner so as to distribute completion messages to worker threads in the order the first queue receives the completion messages;

requesting to service an input/output completion on the queue object by one or more worker threads of an application program, wherein a requesting worker thread blocks if the queue object fails a readiness criterion, and wherein one such readiness criterion is whether the first queue includes a completion message;

on a second of said queues, ordering blocked worker threads for unblocking in a last-in-first-out manner; and

when the queue object includes a completion message, servicing by a worker thread an input/output completion, whereby system resources for worker threads that remain blocked are not immediately needed to process completion messages and thus are

eventually released to promote efficient utilization of system resources.

25. In a computer system with at least one processor, an operating system, and a memory device coupled to the processor and accessible by the operating system, the operating system being a multi-tasking operating system that supports concurrent processing by a plurality of units of execution, a method for synchronizing with completions for asynchronous input/output operations, the method comprising:

requesting plural asynchronous input/output operations;

receiving completion messages for the requested plural asynchronous input/output operations;

waiting on completions for the requested plural asynchronous input/output operations by each of plural units of execution, wherein each of the plural units of execution is capable of processing any of said completions;

maintaining an order for processing said completions; and

processing said completions, whereby maintaining said order reduces variation in times from receiving to processing.

27. The method of claim 25 further comprising:

returning from a function call for waiting on completions, wherein one or more return values include context information for one of the requested plural asynchronous input/output operations.

28. The method of claim 25 wherein maintaining an order for processing said completions includes:

ordering said completion messages in a first-in-first-out completion queue.

30. The method of claim 29 further comprising:

maintaining a last-in-first-out order for the plural units of execution waiting on completions, whereby at least some context data for at least one of said units of execution that remains waiting is eventually swapped to persistent storage to promote efficient memory utilization.

34. The method of claim 25 further comprising:

maintaining a last-in-first-out order for the plural units of execution waiting on completions, whereby at least some system resources for units of execution that remain waiting are eventually released to promote efficient utilization of system resources.

36. In a computer system with at least one processor, a multi-tasking operating system, and a memory device coupled to the processor and accessible by the multi-tasking operating system, the multi-tasking operating system scheduling a plurality of execution entities for concurrent processing, an apparatus for synchronizing execution of plural execution entities with completion of asynchronous input/output operations, the apparatus comprising:

means for accepting requests for one or more asynchronous input/output operations from each of plural execution entities;

input/output manager means for adding completion messages for the asynchronous input/output operations to a synchronization means;

synchronization means for storing said completion messages, said synchronization means ordering processing of completions for the asynchronous input/output operations by plural execution entities that wait upon said synchronization means, whereby said synchronization means provides a single point to synchronize with said completions.

38. The apparatus of claim 36 wherein a completion message includes context information for an asynchronous input/output operation, the apparatus further comprising:

a programming interface means for waiting upon said synchronization means, said programming interface means returning with context information for an asynchronous input/output operation.

40. The apparatus of claim 36 further comprising:

a shared address space that stores return values for the asynchronous input/output operations, wherein each of the plural execution entities that waits upon said synchronization means is able to process a completion for any of the asynchronous input/output operations, whereby the shared address space facilitates processing of completions.

41. The apparatus of claim 36 wherein said synchronization means includes a first-in-first-out completion queue for storing said completion messages.

42. In a computer system with at least one processor, an operating system, and a memory device coupled to the processor and accessible by the operating system, the operating system being a multi-tasking operating system that supports concurrent processing by a plurality of units of execution, a method for synchronizing with completions for asynchronous input/output operations, the method comprising:

requesting plural asynchronous input/output operations;

receiving at a single synchronization point messages for completions for the requested plural asynchronous input/output operations;

waiting at said single synchronization point for said completions by each of plural units of execution, wherein each of the plural units of execution is able to process any of said completions, and wherein a synchronization point indicator checked for waiting units of execution indicates the status of the single synchronization point, thereby avoiding checking of plural completion indicators for the requested plural asynchronous input/output operations for each of the waiting units of execution; and

processing said completions.

44. The method of claim 42 further comprising:

returning from a function call for waiting at said single synchronization point, wherein one or more return values include context information for a completed one of the requested plural asynchronous input/output operations.

45. A computer-readable medium having stored thereon a completion message data structure containing completion information that describes a completed input or output operation for a client having plural units of execution, the completion message data structure comprising:

a first data field containing data representing a quantity of information handled in a completed input or output operation, the completed operation requested by a requesting unit of execution of the plural units of execution;

a second data field containing data representing error indications for the completed operation;

a third data field containing data representing an operation context for the completed operation, the operation context providing operation-specific information usable by a servicing unit of execution of the plural units of execution to service the completed operation; and

a fourth data field containing data representing a client context for the client, the client context providing client-specific information usable by the servicing unit of execution to service the completed operation, whereby the completion message data structure provides completion information independent of the requesting unit of execution and servicing unit of execution.

L48: Entry 22 of 30 File: USPT Oct 27, 1998

DOCUMENT-IDENTIFIER: US 5828881 A
TITLE: System and method for stack-based processing of multiple real-time audio tasks

Brief Summary Text (12):
In accordance with an embodiment of the present invention, a software system is provided which controls a real-time signal processor. The real-time signal processor includes a host computer, a message queue, a stack and a media I/O circuit. The host computer sends a plurality of messages designating ones of a plurality of corresponding signal processing operations asynchronously to the message queue. Each signal processing operation corresponds to a client processing module. The software system includes a message processing routine for reading the messages from the message queue and dispatching the messages to designated ones of the plurality of client processing modules and a process sequencing routine for invoking client processing modules and for controlling the stack. The client processing modules communicate signals as a sending client processing module pushes a signal onto the stack and a receiving client processing module pops the signal from the stack.

Brief Summary Text (13):
In accordance with another embodiment of the invention, a signal processing system includes a host computer system having a host processor and an execution memory, a media input/output circuit connected to the host computer system. The media input/output circuit includes a media signal processor and a memory connected to the media signal processor. The memory includes a stack. The signal processing system further includes a host operating system for operating on the host processor, a host task for operating under the host operating system on the host processor and generating a net list of messages on a message queue for selecting signal processing processes, a resource manager (RM) software subsystem for operating on the host computer system and an interprocess communication operating system (XOS) for interacting in cooperation on the host processor and the media signal processor. The signal processing system also includes a synchronous net list processing routine for executing under XOS on the media signal processor. The synchronous net list processing routine has a plurality of client processing modules each corresponding to a signal processing process. Each client processor module processes signals and communicates signals to a client processor module. The synchronous net list processing routine also includes a message processing routine for reading the messages from the message queue and synchronously dispatching the messages to designated ones of the plurality of client processing modules and a process sequencing routine for invoking client processing modules and for controlling the stack. The client processing modules communicate signals in which a sending client processing module pushes a signal onto the stack and a receiving client processing module pops the signal from the stack.

Brief Summary Text (14):
In accordance with a further embodiment of the invention, a method of scheduling a plurality of signal processes designated by a net list of signal processes uses a stack-based protocol to manage signal flow topology. The method includes the steps of generating a plurality of sequential output synchronization timing signals, parsing a new net list of signal processes to generate a new signal processing

sequence, and writing the signal processing sequence into a message queue. The method further includes the steps of waiting for an output synchronization timing signal and, in response to an output synchronization timing signal, reading the signal processing sequence from the message queue. The method then copies the signal processing sequence into a process sequence table so that a new signal flow topology is specified.

Brief Summary Text (15):
Many advantages are achieved by the disclosed system and method. One advantage is that signal flow is managed so that various processes function essentially simultaneously in a dynamically configurable topology of signal flow so that system resource usage is conserved. Another advantage is that usage of stack-based signals to communicate between real-time tasks advantageously smoothes playback of audio signals while furnishing accurate waveform sampling and reproduction. It is further advantageous that messages are listed relative to one another in the timed message queue so that timing of the multiple messages is synchronized and a separate accurate timer is not necessary. Usage of the stack for dynamically configurable topology of signal flow handling is advantageous since memory usage is reduced in comparison to usage of dedicated memory elements. Because the amount of memory used is reduced, the stack is advantageously implemented in static RAM so that speed performance is enhanced. Usage of the stack allows the SRAM to be shared between many devices.

Detailed Description Text (13):
RM 222 controls static resource allocation and I/O bandwidth in memory circuit 120 by controlling thread processing. A thread is a collection of processes having an order which determines which process is eligible for execution. A thread is an element that is scheduled. Resources such as execution time, locks and queues are assigned to a thread. RM 222 controls thread processing first by allocating memory space in DRAM 126, then by creating and evicting threads. RM 222 allocates both instruction code and data sections of the DRAM 126. First, RM 222 allocates a DRAM 126 code section for creation of a thread, then links and downloads code into DRAM 126 blocks while a thread is active. RM 222 avoids improper access to the code segment of a thread by temporarily blocking the thread while relocating the code segment. Each XOS thread has one entry point that is called when the thread is created by RM 222. RM 222 then allocates a data space in DRAM 126 for the thread. A thread may own multiple data segments in DRAM 126 and switches between these segments. Only RM 222 can increase the DRAM 126 allocation of a thread.

Detailed Description Text (15):
Once RM 222 creates a thread, a device driver determines when the thread accesses the media operating system 220 by activating the media operating system 220 through operations of the RM 222. After RM 222 allocates space for a thread, RM 222 creates and places the thread in a non-scheduled, "paused" state until initialization of the thread is finished. When the thread is ready to execute, RM 222 changes the thread state to "active" so that the thread is scheduled for processing. RM 222 selectively and asynchronously pauses a thread at any time, thereby rendering the thread inactive. RM 222 also selectively reactivates the thread later. RM 222 usually asynchronously pauses a thread before releasing (deallocating) the thread.

Detailed Description Text (18):
XOS 224 is a collection of media operating system-based software that enables the real-time multitasking and interprocess communication on the media operating system 220. XOS 224 controls use of the on-chip static RAM 140 through the control of threads. For each thread, XOS 224 specifies a data area and instruction cache location of the on-chip static RAM 140. XOS 224 also controls the data or code segments of DRAM 126 that the thread currently uses. XOS 224 supports queues for passing messages and data between media operating system 220 and threads and software tasks running under the host operating system 210.

Detailed Description Text (19):
A thread, an interrupt service routine (ISR), or software running under the host operating system 210 uses RM 222 to post a queued message to an XOS thread. A thread or software routine executing under the host operating system 210 using RM 222 waits for a queued message from an XOS thread. XOS queues are located in DRAM 126.

Detailed Description Text (21):
Active threads execute in one of three states. The states include a running state when the thread is executing under the media operating system 220, a waiting state when the thread is unable to run until an event occurs and a read state while the thread is halted to allow another thread having a nearer deadline to execute. For each thread, XOS 224 specifies the on-chip static RAM 140 location of target data areas and instruction cache areas. RM 222 allocates DRAM 126 space, and creates and evicts threads. Once RM 222 creates a thread, the appropriate device driver of device drivers 202, 204 and 206 determines when the thread gains access to the media operating system 220 by activating the thread through RM 222. XOS 224 controls which DRAM 126 data or code segments the thread currently uses.

Detailed Description Text (22):
A thread having the closest deadline runs. Only one thread runs at any instant. A thread starts running in one of two ways. If a waiting thread receives a posted event, the thread starts running ready if its deadline is the closest. A thread that is ready starts running when its deadline suddenly becomes the closest because the previously running thread has called an XOS wait function or has finished processing. A thread stops running in one of two ways. When the running thread calls an XOS wait function, it changes to the waiting state. When another thread's deadline becomes the closest, the thread currently running is moved to a ready state until its deadline is once again the closest.

Detailed Description Text (23):
A thread begins waiting by calling an XOS wait function. When a waiting thread receives a Post from an interprocess communication (IPC), the thread begins running if the deadline is the nearest. If a thread receives a posted message, but the deadline is not nearest of other ready threads, the thread changes to a ready state. A thread becomes ready in one of two ways. If a running thread is preempted by another thread with a nearer deadline, the preempted thread changes to ready until the deadline is the nearest. If a waiting thread receives a posted message, but the deadline is not nearest, the thread changes from waiting to ready. A thread changes from ready to running when the deadline is nearest.

Detailed Description Text (30):
In set task state to paused step 316, RM 222 places the thread in a nonscheduled, "paused" state until initialization of the thread is complete. When the thread is ready to run, RM 222 changes the thread state to "active" state so that the thread is scheduled for running. After allocating space for the thread, RM 222 can asynchronously pause a thread at any time, rendering the thread inactive. RM 222 can subsequently reactivate the thread.

Detailed Description Text (36):
Referring to FIG. 5, a state diagram showing operative states of a thread illustrates that both interrupt service routines (ISRs) and XOS threads post interprocess communication (IPC) messages. Tasks operating under the host operating system 210 post an IPC message through RM 222. When the deadline of a thread becomes the nearest deadline, XOS 224 automatically preempts a currently running thread. The preempted thread enters the ready state and remains in a ready state until the deadline of the preempted thread is once again the nearest. Accordingly, XOS 224 supports real-time preemptive threads with nearest deadline scheduling. A running thread is preempted by another thread only when several conditions are satisfied. First, an interrupt occurs and the ISR corresponding to the interrupt

posts an IPC event to one or more threads. Second, at least one of the posted threads is waiting for the ISR's IPC event. Third, at least one of the posted threads has a nearer deadline than the currently running thread.

Detailed Description Text (37):
A host task utilizes RM 222 to call an RM function that posts a thread signal. Similarly, a task awaits a return from RM 222 or awaits a post to RM 222 using an RM function. Typically, tasks running under the host operating system 210 do not wait for a signal from an XOS thread.

Detailed Description Text (41):
FIG. 7 is a flow chart illustrating an overview of the functionality of a synchronous net list processing routine 700. A task running under the host operating system 210 sends messages asynchronously and posts these messages to two first-in-first-out (FIFO) message queues located in DRAM 126 shown in FIG. 1. The two message queues are an untimed message queue and a timed message queue, that are used by the synchronous net list processing routine 700. Synchronous net list processing routine 700 functions in an event-driven loop and typically begins the loop in a waiting state, shown in FIG. 7 by a wait for activation signal step 710. Synchronous net list processing routine 700 waits for a thread signal from the PBUS 138 interrupt service routine, which occurs when an audio signal coder/decoder (codec) requires more samples. After synchronous net list processing routine 700 is activated by the interrupt, synchronous net list processing routine 700 reads messages from the message queues. Synchronous net list processing routine 700 first reads messages from the untimed message queue in read untamed messages step 712. Untimed messages are read in the order of reception. Synchronous net list processing routine 700 then dispatches pending messages from the untimed message queue to a process table which designates message handling routines that are associated with corresponding client processes, in dispatch untimed messages step 714. The process table is an array of process slots, each of which holds information for referencing and calling a subprocess of the synchronous net list processing routine 700. Following dispatch untamed messages step 714 control loops back to read untimed messages step 712 until all untimed messages are read.

Detailed Description Text (42):
Next, in read timed messages step 716, synchronous net list processing routine 700 reads timed messages in a serial manner, checking the time stamp of the first timed message in check time stamp step 718. If a time stamp exceeds current time, as determined by synchronous net list processing routine 700, the message is not processed and left in the timed message queue. Otherwise, timed messages are dispatched to the process table in dispatch timed messages step 720. Timed messages are dispatched in chronological sequence so that only the first message is checked for the time stamp. If the first message is left in the timed message queue, subsequent messages have an equal or greater time stamp so that all messages are left in the queue. The timed message queue allows precise sequencing of control messages to synchronous net list processing routine 700 client processes via message handling routines of processes.

Detailed Description Text (46):
The synchronous net list processing initialization module 810 is called when the media processor system 100 first begins execution. Substantially all software and hardware structures that are utilized in synchronous net list processing are initialized in the initialization module 810 so that initialization is consolidated without mixing state contexts of the various structures. Initialization sets up the synchronous net list processing routine 800 to queue information from RM 222 in a timed message queue and an untimed message queue and to receive semaphore information from RM 222. Usage of a timed message queue advantageously sequences playback of audio signals while furnishing accurate waveform sampling and reproduction. Because messages are posted with precise time stamps in the timed message queue, timing of the multiple messages are synchronized so that an accurate

timer is not necessary. Set up process table step 812 sets initial operating parameters of a process table in DRAM 126. Global initialization step 814 sets up interrupt drivers such as a coded signal MIDI interrupt driver (not shown). Global initialization step 814 also initializes various parameters in the stack and acquires an identifier for each queue of a plurality of queues. A stack initialization step 818 initializes the stack including setting the stack pointer. A process manager initialization step 820 initializes the number of active processes to zero. A timebase initialization step 822 sets a time reference to zero. The timebase is used for comparing to a time stamp of a message to determine whether to perform a message on the timed message queue. A clear peripheral bus step 824 clears the peripheral bus (PBUS) buffers prior to starting the peripheral bus. An initialize peripheral bus step 826 sets the peripheral bus to begin data transfers.

Detailed Description Text (47):
The main operating module 840 includes library code for the synchronous net list processing routine 800 and defines data structures in DRAM 126. Main operating module 840 furnishes an operating framework for processing messages that are received in real time. Main operating module 840 operates in a main event loop within which all message processing operations of the synchronous net list processing routine 800 take place. Timing is controlled in the media processor system 100 on the basis that the main operating module 840 operates in a loop which is based on externally timed signals. The main operating module 840 begins with a loop entry point 842 at which a wait step 844 is positioned. In wait step 844, software waits for audio codec output synchronization step or an immediate signal using a thread signal handler which is set to process immediate signals. A message with a properly formatted header is posted to an appropriate queue in XOS 224, either the timed message queue or the untimed message queue. When a signal is received, at entry point 846, software checks incoming message queues in check queues step 848. Check queues step 848 reads the headers of the messages and tests that the entire length of the message body is loaded into the queue buffer. Check queues step 848 first checks the untimed message queue and then checks the timed message queue. If messages are available to be processed, dispatch messages step 850 performs message processing by forwarding the designated messages to message recipient routines shown as a pluralilty of message handling routines 860. The recipient routine is simply a target function. If the recipient's indicated context data is in relocatable memory space, the data segment register is set using a designated handle and the indicated message handler function is called. If the code is in relocatable space, which is designated by a cleared sign bit, then a far call operation is performed. Otherwise a normal call is made.

Detailed Description Text (48):
In dispatch messages step 850, entries from the timed and untimed message queues are read into a message buffer, which is located in DRAM 126 shown in FIG. 1. Up to 64 double words of messages are read from the queue at one time. Batching of message queue read operations reduces overhead of XOS queue accesses. As many untimed messages as fit in the message buffer are repeatedly read from the untimed message queue until the queue is empty. Timed messages are read from the timed message queue and buffered locally. Timed messages are processed in the order of reception and wait until the current system time, measured in ticks, is equal to or greater than the time stamp of the oldest message. The recipient routines set up the process table.

Detailed Description Text (50):
After processing of each message is complete, the dispatch messages step 850 loops back to the check queues step 848. In loop through process table step 852, software progresses through the process table in a round-robin order, executing each module in sequence. When all modules of the process table have been executed, clear thread signal step 854 clears the activating thread signal and update timing step 856 updates the tick timer and then loops to the loop entry point 842 where software

waits for audio codec interrupt before continuing.

Detailed Description Text (51):
The dispatch messages step 850 dispatches messages to a message handling routine of message handling routines 860, which returns at completion. Messages are variable in length and include a three double word header followed by data. RM 222 passes messages into the untimed or timed message queue of the synchronous net list processing routine 800 using XOS 224 data structures and communication routines. Messages from the timed message queue that have been read but not processed are transferred to a DRAM 126 storage area to temporary holding. The message module contain routines that handle storage and retrieval of timed messages from the temporary storage area in DRAM 126. The dispatch messages step 850 is called with various parameters set. A current queue identifier is set to a designated XOS queue, either the untimed message queue or the timed message queue. A message read byte remaining count is set to the number of bytes left in the queue buffer which is currently read. A read pointer is set to the beginning of the valid portion of the queue buffer from which reading is to begin. The read pointer may be set to zero to reset the pointer to the beginning of the queue buffer. The queue buffer, either the untimed or the timed queue buffer, is preloaded with appropriate data in appropriate locations if the read pointer is not set to zero. A timed message queue flag is set if timed message processing is desired. For timed messages, the dispatch messages step 850 dispatches messages while comparing time stamps with the current tick count. Dispatch messages step 850 returns when no additional messages are held on the timed message queue or until a message with a future time stamp is encountered. The unprocessed message is retained in the timed message queue buffer. For a retained message, the calling function that called the retained message saves the timed message queue buffer in temporary storage in DRAM 126 and restores the message to the timed buffer queue in the next call. The read pointer and the read byte remaining count reflect the status of the queue buffer.

Detailed Description Text (52):
If the timed message queue flag is not set then the queue to be read is the untimed message queue. For a message in the untimed message queue, the queue is read until empty. The time stamp field is ignored. Upon return from the dispatch messages step 850 following processing of an untimed message, the queue buffer is assumed to be empty.

Detailed Description Text (53):
Referring to FIG. 9, a flow chart shows steps of the dispatch messages step 850. In a message processing initialization step 862, software resets the read pointer to the beginning of the queue buffer and writes a peak queue buffer load level value to cache for later comparison. Software also saves a return link value. Load message queue buffer step 864, software normalizes data for a double word count rather than for a byte count and saves a link register. Load message queue buffer step 864 then compacts the message buffer toward the front of the buffer, calculates the base address of the buffer and resets the buffer pointer to the beginning of the buffer. Parse message from buffer step 866 reduces the constituent fields of the message in preparation for dispatching of the message. A timed message logic step 868 determines whether the message is a timed message or an untimed message. If the message is a timed message, check time stamp step 870 determines whether timing of the timed message is correct. If not, dispatch messages step 850 returns. If the timing is correct, dispatch message to handler step 872 dispatches parsed messages to the process table. If the message is an untimed message, as determined by check time stamp step 870, dispatch message to handler step 872 is performed without timing checks. Dispatch message to handler step 872 dispatches messages to a message handling routine of message handling routines 860. Check for more messages step 874 checks for more messages and loops to parse message from buffer step 866 when more messages are in the queue and otherwise returns.

Detailed Description Text (54):
Referring again to FIG. 8, the client process routines 890 include a plurality of signal processing modules that are designated by the process table. A control block in the synchronous net list processing routine 800 has data fields for tracking the number of total processes to be run and the next process to run. To produce one of sound samples, usually 64 samples, client process routines are activated by progressing down the process table, loading the next process slot from a process slot table in DRAM 126 and performing a call or far call operation to the registered entry point of a subprocess. A client process may alter its entry point for the next call or may cause synchronous net list processing routine 800 to skip a number of entries on the queue by modifying a next process pointer. This characteristic is useful for a process that acts as a gate for dependent subprocesses, allowing a block to execute conditionally.

Detailed Description Text (55):
A structure is defined which contains macro, typedef, memory map and other definitions that are used by a client routine running under the synchronous net list processing routine 800. This structure is organized to include temps, constants and states. Temps are used as scratchpad space during tick computation. Constants include control parameters and constants that are read-only with respect to the tick computation loop. Constants are often subdivided into private constants and constants which are shared among more than one instantiation of the module. Constants define operational parameters such as stack dimensions, time tick duration, data sample sizes, queue buffer sizes, buffer base addresses, and maximum message size. Other constants designate interface parameters such as audio codec input and output channels, peripheral bus offset definitions, and the like. A state is defined by variables that are altered by tick computation and preserved for subsequent iterations.

CLAIMS:

1. A software system for controlling a real-time signal processor, the real-time signal processor including a host computer, a message queue, a stack and a media I/O circuit, the host computer for sending a plurality of messages designating ones of a plurality of corresponding signal processing operations asynchronously to the message queue, each signal processing operation corresponding to a client processing module wherein:

the media I/O circuit includes a means for generating a plurality of sequential output synchronization timing signals;

the host computer includes a host task having subroutine for parsing a new net list of signal processes to generate a new signal processing sequence;

the host task includes a subroutine for writing the signal processing sequence into a message queue;

the software system comprising:

a message processing routine for reading the messages from the message queue and dispatching the messages to designated ones of the plurality of client processing modules;

a process sequencing routine for invoking client processing modules and for controlling the stack, the client processing modules communicating signals by a sending client processing module pushing a signal onto the stack, and a receiving client processing module popping the signal from the stack;

a subroutine for waiting for an output synchronization timing signal;

a subroutine operating in response to an output synchronization timing signal for reading the signal processing sequence from the message queue; and

a subroutine for copying the signal processing sequence into a process sequence table so that a new signal flow topology is specified.

4. The software system according to claim 1 wherein:

the message queue includes a timed message queue on which timed messages are queued, the timed messages being encoded with a time stamp; and

the message processing routine dispatches messages synchronously by comparing the time stamp to a current time counter, dispatching messages when due as determined by the comparison and holding messages that are not due for later dispatch.

5. The software system according to claim 1 wherein:

the message queue includes an untimed message queue on which untimed messages are queued; and

the message processing routine dispatches untimed messages when untimed messages are pending.

11. A software system for controlling a real-time signal processor, the real-time signal processor including a host computer, a message queue, a stack and a media I/O circuit the host computer for sending a plurality of messages designating ones of a plurality of corresponding signal processing operations asynchronously to the message queue wherein:

the media I/O circuit includes a means for generating a plurality of sequential output synchronization timing signals;

the host computer includes a host task having a subroutine for parsing a new net list of signal processes to generate a new signal processing sequence;

the host task includes a subroutine for waiting the signal processing sequence into a message queue;

the software system comprising:

a plurality of client processing modules, each client processor module for processing signals and for communicating signals to a client processor module;

a message processing routine for reading the messages from the message queue and synchronously dispatching the messages to designated ones of the plurality of client processing modules;

a process sequencing routine for invoking client processing modules and for controlling the stack, the client processing modules communicating signals by a sending client processing module pushing a signal onto the stack, and a receiving client processing module popping the signal from the stack;

a subroutine for waiting for an output synchronization timing signal;

a subroutine operating in response to an output synchronization timing signal for reading the signal processing sequence from the message queue; and

a subroutine for copying the signal processing sequence into a process sequence table so that a new signal flow topology is specified.

16. A signal processing system comprising:

a host computer system including a host processor and an execution memory;

a media input/output circuit coupled to the host computer system and including a media signal processor and a memory coupled to the media signal processor, the memory including a stack;

a host operating system for operating on the host processor;

a host task for operating under the host operating system on the host processor and generating a net list of messages on a message queue for selecting signal processing processes;

a resource manager (RM) software subsystem for operating on the host computer system;

an interprocess communication operating system (XOS) for interacting in cooperation on the host processor and the media signal processor;

a synchronous net list processing routine for executing under XOS on the media signal processor, the synchronous net list processing routine further including:

a plurality of client processing modules each corresponding to a signal processing process, each client processor module for processing signals and for communicating signals to a client processor module;

a message processing routine for reading the messages from the message queue and synchronously dispatching the messages to designated ones of the plurality of client processing modules; and

a process sequencing routine for invoking client processing modules and for controlling the stack, the client processing modules communicating signals by a sending client processing module pushing a signal onto the stack, and a receiving client processing module popping the signal from the stack.

17. A signal processing system according to claim 16, wherein:

the media signal processor includes a codec for generating a plurality of sequential output synchronization timing signals;

the host task includes a subroutine for parsing a new net list of signal processes to generate a new signal processing sequence;

the host task includes a subroutine for writing the signal processing sequence into a message queue; and

the synchronous net list processing routine further includes:

a subroutine for waiting for an output synchronization timing signal;

a subroutine operating in response to an output synchronization timing signal for reading the signal processing sequence from the message queue;

a subroutine for copying the signal processing sequence into a process sequence table so that a new signal flow topology is specified.

19. A method of scheduling a plurality of signal processes designated by a net list of signal processes using a stack-based protocol, the method comprising the steps of:

generating a plurality of sequential output synchronization timing signals;

parsing a new net list of signal processes to generate a new signal processing sequence;

writing the signal processing sequence into a message queue;

waiting for an output synchronization timing signal;

in response to an output synchronization timing signal, reading the signal processing sequence from the message queue; and

copying the signal processing sequence into a process sequence table so that a new signal flow topology is specified.